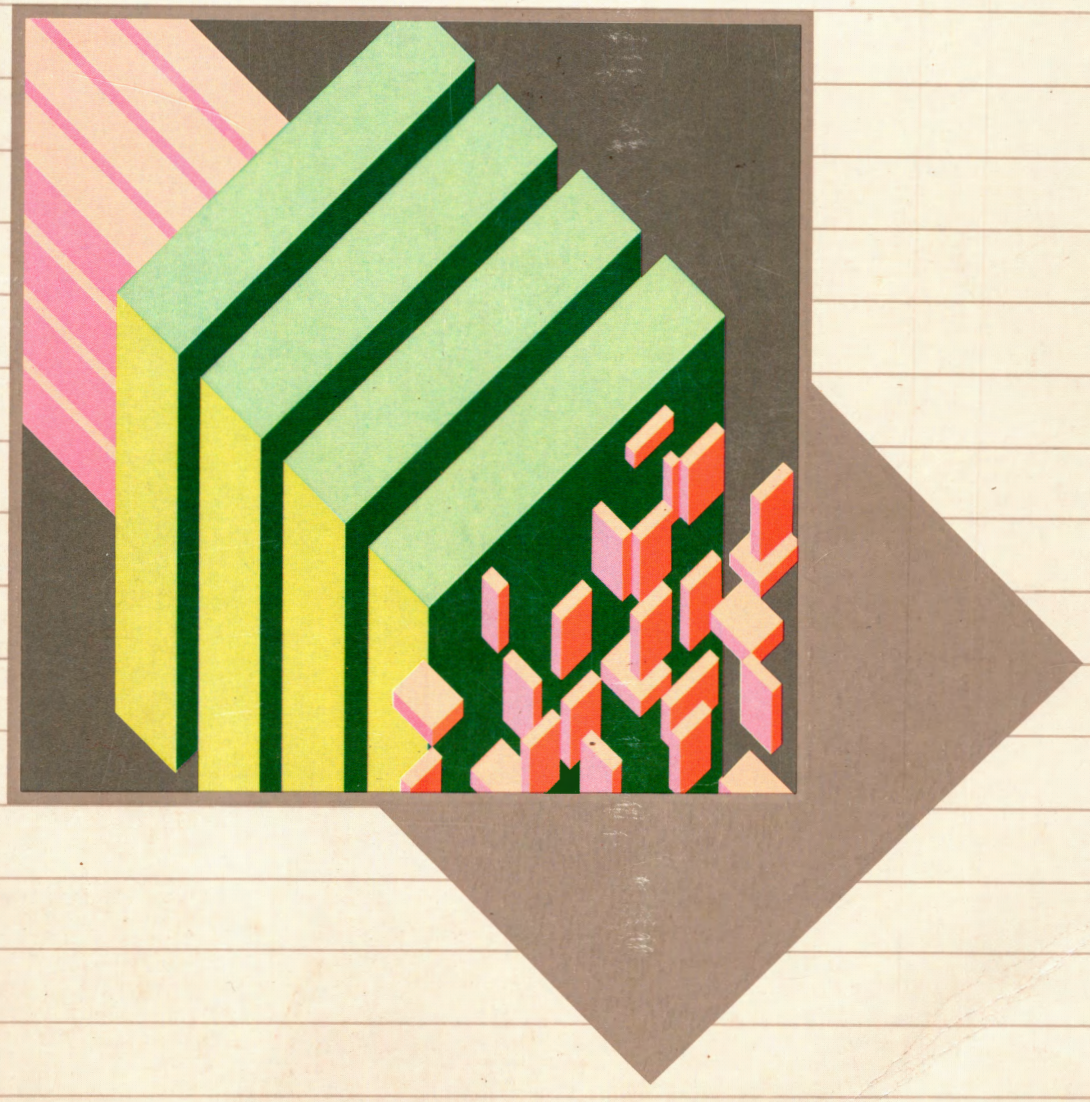
Apple //

Applesoft BASIC Programmer's Reference Manual - Volume I

For //e Only



Notice

Apple Computer, Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

Disclaimer of All Warranties and Liabilities

Apple Computer, Inc. makes no warranties, either express or implied, with respect to this manual or with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Apple Computer, Inc. software is sold or licensed "as is." The entire risk as to its quality and performance is with the buyer. Should the programs prove defective following their purchase, the buyer (and not Apple Computer, Inc., its distributor, or its retailer) assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Apple Computer, Inc. be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the software, even if Apple Computer, Inc. has been advised of the possiblity of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

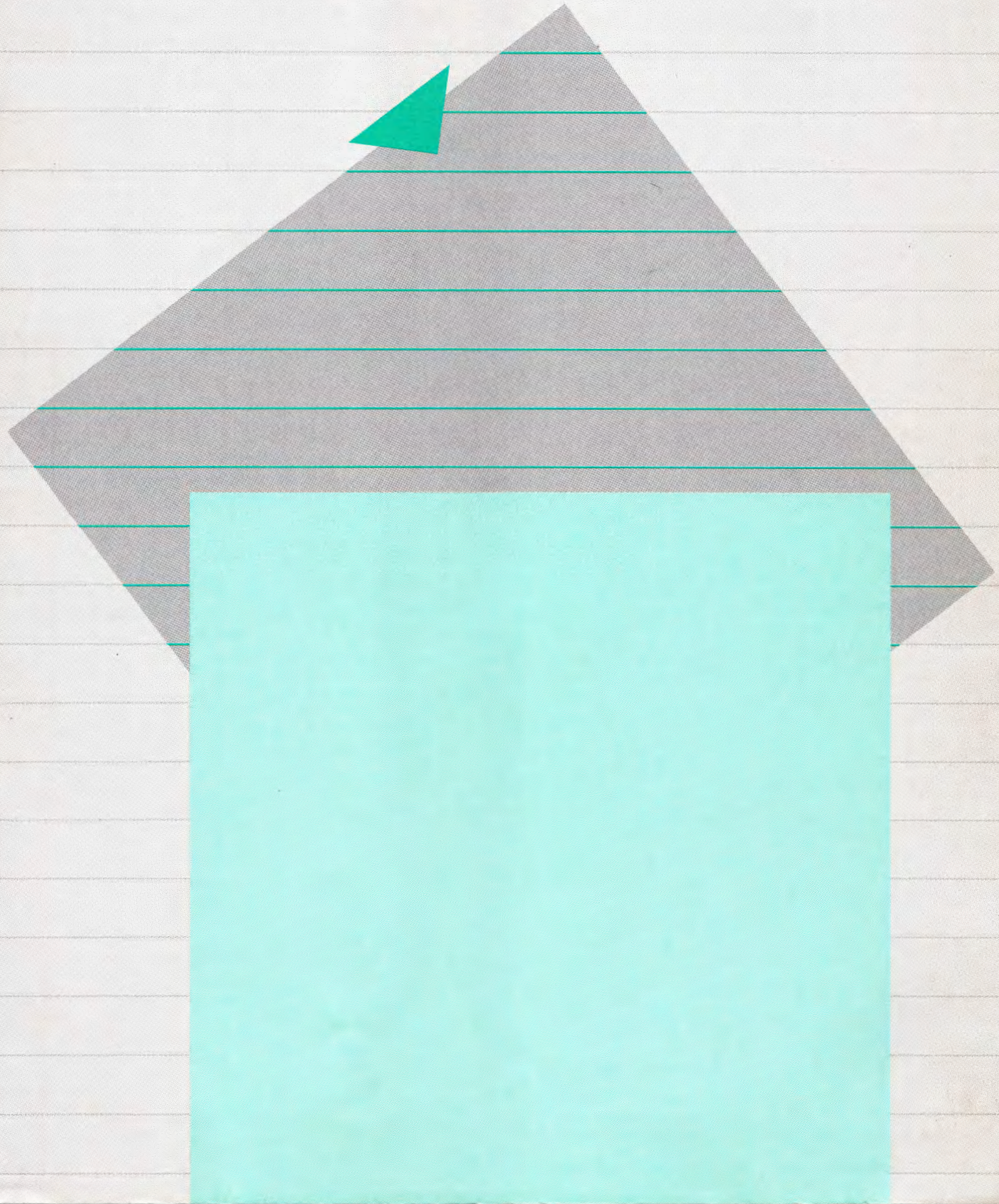
Simultaneously published in the U.S.A and Canada.

This manual was written for Apple Computer, Inc., by
Scot Kamins
Technology Translated
San Francisco, California

Reorder Apple Product #A2L2005

Apple *II*

Applesoft BASIC Programmer's Reference Manual - Volume 1

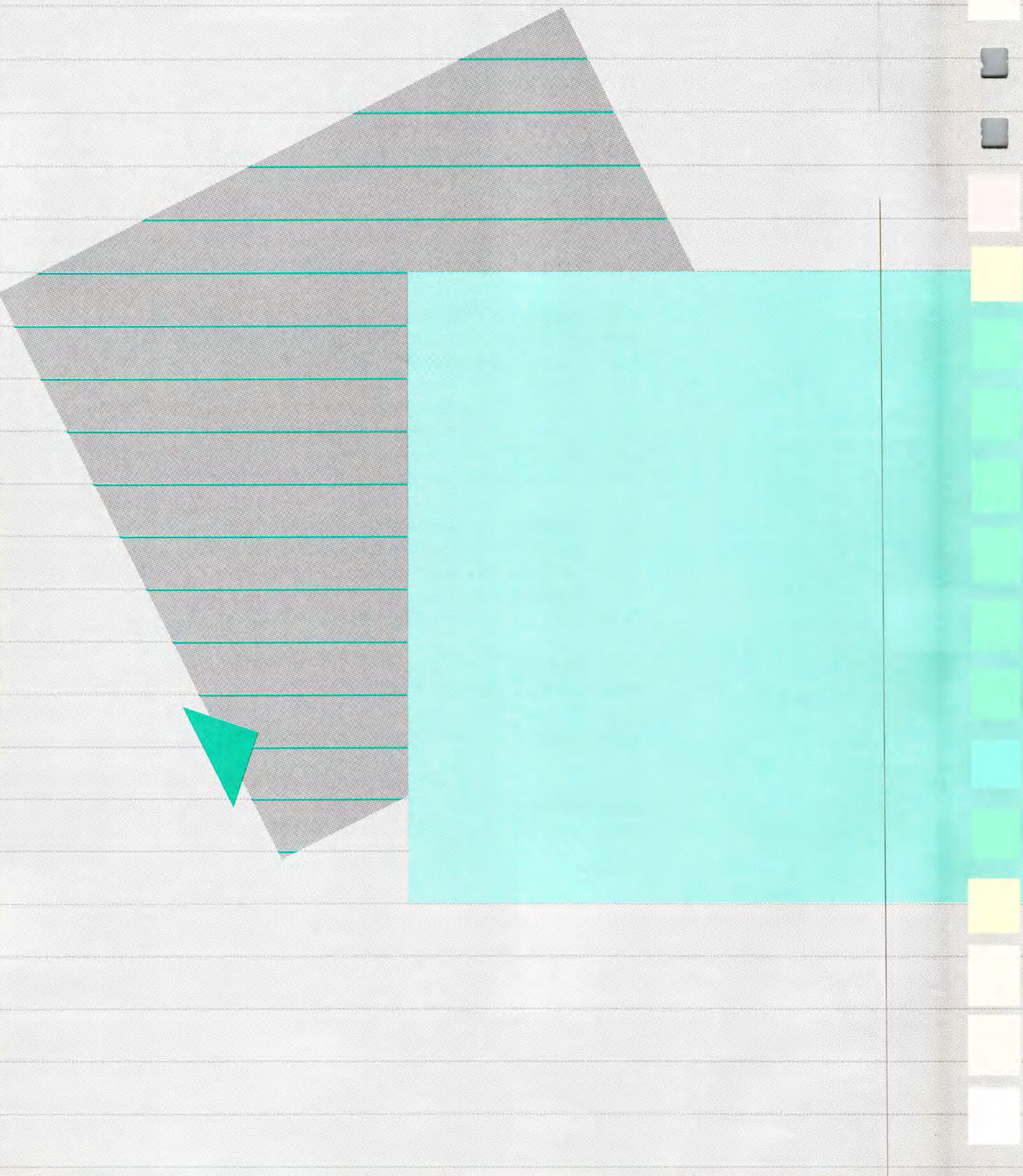


Volume One

Volume Two

List of Figures

List of Tables



About This Manual

This is a reference manual for the Applesoft BASIC programming language as implemented on the Apple IIe computer. It is intended for readers who have had some previous experience with programming, either in BASIC or in some other programming language. It assumes that you are familiar with the material in the *Apple IIe Owner's Manual,* and if you are a novice programmer, that you have read the Apple IIe *Applesoft Tutorial.*

To make using this manual easier for you, we have divided it into two volumes. The complete table of contents, chapters one through eight, and the complete index appear in volume one, the volume you are now reading. Volume two holds the appendices and the glossary; the index is also included in this volume for your convenience.

Purposes of This Manual

This manual has four purposes:

- To serve as a complete reference manual to the Applesoft BASIC language for the experienced programmer.
- To provide clear enough explanations and examples so that a new programmer can learn the details of any statement quickly and easily.
- To allow any reader, even one who is not trying to learn Applesoft in detail, to get a general feel for the language.
- To provide an introduction to program planning, design, and development for the programmer-in-training.

This manual is decidedly not a tutorial. Experienced programmers can learn a great deal about Applesoft by reading it from the first page straight through to the end; but it wasn't designed to be used in that way.

Where to Learn More

The following sources contain further information about the Apple IIe computer in general and the Applesoft programming language in particular:

- The *Apple IIe Owner's Manual* covers the basics of the system and includes a special section on the Apple IIe's keyboard. It also contains a list of books and magazines of special interest to Applesoft programmers, as well as a short guide to the rest of the extensive documentation that comes with your Apple IIe.
- APPLE PRESENTS...APPLE is a training disk that comes with all disk-based Apple IIe systems. It contains an interactive tutorial program giving you hands-on practice with many of the concepts discussed in the *Apple IIe Owner's Manual.* It's a must if you're new to computers.
- The Apple IIe *Applesoft Tutorial* is an excellent guide for beginning programmers. It provides introductory, step-by-step guidance for the new programmer and has a special chapter on editing Applesoft programs.
- *Apple Backpack: Humanized Programming in BASIC,* by Scot Kamins and Mitchell Waite (BYTE/McGraw-Hill Books) fills the gap some newer programmers may feel between the Applesoft Tutorial and this reference manual. It teaches programming in a friendly and easy-paced way for people who are not computer experts.
- The *Apple IIe Reference Manual* contains a wealth of information about the more technical aspects of the system's operation, with lists of various programmer-accessible system flags, pointers, and soft switches.

How This Manual Is Organized

This manual has 8 chapters, 14 appendices, a glossary of terms, an index, and a quick reference card. All of it is designed to help you get the most out of Applesoft. Here's a description of what each chapter and appendix is about:

Chapter 1, "General Information," contains information every Applesoft programmer needs. It discusses the programming environment in which Applesoft operates and tells how to create, modify, execute, and store Applesoft programs.

Chapter 2, "Variables and Arithmetic," deals with some of the most fundamental concepts of Applesoft programming: variables, arithmetic expressions and operators, arithmetic precedence, Applesoft's built-in functions, and how to define and use your own functions.

Chapter 3, "Control," covers the various statements available to direct the flow of program execution. It includes information on unconditional and conditional branching, loops, subroutines, error handling, and program termination.

Chapter 4, "Arrays and Strings," completes the material on variables begun in Chapter 2. It includes information on the definition and use of arrays in Applesoft and on the various string manipulation facilities.

Chapter 5, "Input/Output," describes Applesoft's facilities for getting information into and out of programs and for formatting the way information is presented on the display screen.

Chapter 6, "Graphics," tells how to create, change, display, and store low- and high-resolution graphic designs. There is an extensive discussion on creating and using shape tables, as well as examples of how to create animation sequences.

Chapter 7, "Utility Statements," contains information on a variety of miscellaneous Applesoft facilities for low-level control of the programming environment: directly accessing specific memory locations, controlling the limits of program space, and tracing the execution of a program for debugging purposes.

Chapter 8, "Bringing It All Together," is more tutorial than any other chapter in the manual; it describes and demonstrates a method for planning, designing, and developing efficient, bug-free (well, *relatively* bug-free) programs.

Appendix A, "Summary of Applesoft Statements and Functions," gives an abbreviated description of each Applesoft statement and function, together with a reference to the chapter, section, or appendix where you can find more detailed information and examples.

Appendix B, "Syntax Definitions," defines terms used in the formal syntactic definitions of Applesoft statements given in Appendix A. In the body of the manual, statement syntax is shown by example rather than by formal definition; most readers can safely avoid the formal definitions altogether.

Appendix C, "ASCII Character Codes," contains a complete listing of the ASCII characters; it is an adjunct to the comments on ASCII in chapter 4.

Appendix D, "Reserved Words," is a list of words (some of them rather odd-looking) that cannot be used in variable names.

Appendix E, "Error Messages," describes the meanings of the error messages that Applesoft displays on the screen. Each description includes an explanation of why the error occured; in some cases, there are suggestions for debugging.

Appendix F, "Peeks, Pokes, and Calls," deals with low-level access to features of the Apple IIe computer via Applesoft's PEEK function and POKE and CALL statements. There are sections on screen text, the keyboard, graphics, miscellaneous input and output, and error handling.

Appendix G, "Hints for Program Efficiency," offers techniques for cutting down the size of programs and for speeding up program execution.

Appendix H, "Implementation Details," contains information of interest mainly to the advanced programmer. Included here is a memory map with a list of pointers and their descriptions, information on Applesoft's methods of internal storage allocation, an outline of its usage of special locations in page 0 of memory, and a list of the tokens it uses for internal representation of keywords.

Appendix I, "Display Formats for Numbers," describes how Applesoft displays numbers on the screen and gives the ranges of numbers the system is capable of handling.

Appendix J, "On-Screen Editing and Cursor Control," contains tables summarizing Applesoft's on-screen editing features.

Appendix K, "40/80-Column Differences," is a table showing the differences in Applesoft's behavior with and without the optional Apple IIe 80-Column Text Card installed.

Appendix L, "Comparison with Integer BASIC," gives charts showing the differences between Applesoft and Apple Integer BASIC and discusses how to convert Integer BASIC and non-Apple-IIe BASIC programs into Applesoft.

Appendix M, "If You Have a Cassette Recorder," describes Applesoft's statements for using tape cassettes as a storage medium for programs and information.

Appendix N, "Complete Listing of the Postage Rates Program," gives the complete text of the programming example developed in Chapter 8.

At the back of the manual is a Glossary of technical terms. In general, buzz words are no-no's in this manual; but any technical field has its own jargon, developed out of necessity to describe concepts genuinely having no parallel in common language. The glossary lists all (we hope!) such words and terms that have found their way into the manual, and a few others besides.

A tear-out Quick Reference Card, designed to act as a "memory jog," gives an extremely brief description of each statement, function, operator, and variable type.

How to Use This Manual

Here are some suggestions on how to use this manual, depending on the particular goals you are trying to accomplish.

As a Reference

- Look up the feature of interest on the Quick Reference Card; each statement, function, operator, and variable type is listed there in an extremely abbreviated form as a "memory jog."
- Look up the feature in Appendix A, "Summary of Applesoft Statements and Functions"; each statement and function is described briefly, and a reference is given to the chapter, section, or appendix where it is discussed in detail.
- Look up the feature in the index; there you'll find references to the places in the manual where it is mentioned.
- Look in the appendices at the back of the manual for quick reference on specific facts.

To Learn the Applesoft Language

- Read Appendix A, "Summary of Statements and Functions," to get a quick feel for each of the features in the language.
- Read through each chapter and enter and run the example programs; then try modifying them to check your understanding and gain hands-on experience.
- Enter, run, and modify the example program in chapter 8.

To Learn Program Planning

- Read through chapter 8 and experiment with the program developed there.
- Develop your own programs based on the methods presented in chapter 8.
- Restructure someone else's program using the methods of chapter 8.
- Read Appendix G, "Hints for Program Efficiency," at the back of the manual.

Conventions Used in This Manual

Throughout this manual you'll encounter the following conventions:

Warning
Warning boxes contain vital information about potentially dangerous situations in which you can damage or destroy equipment, programs, or information.

Grey boxes contain minor details, tricky points, side comments, helpful hints, historical notes, and other information of secondary importance.

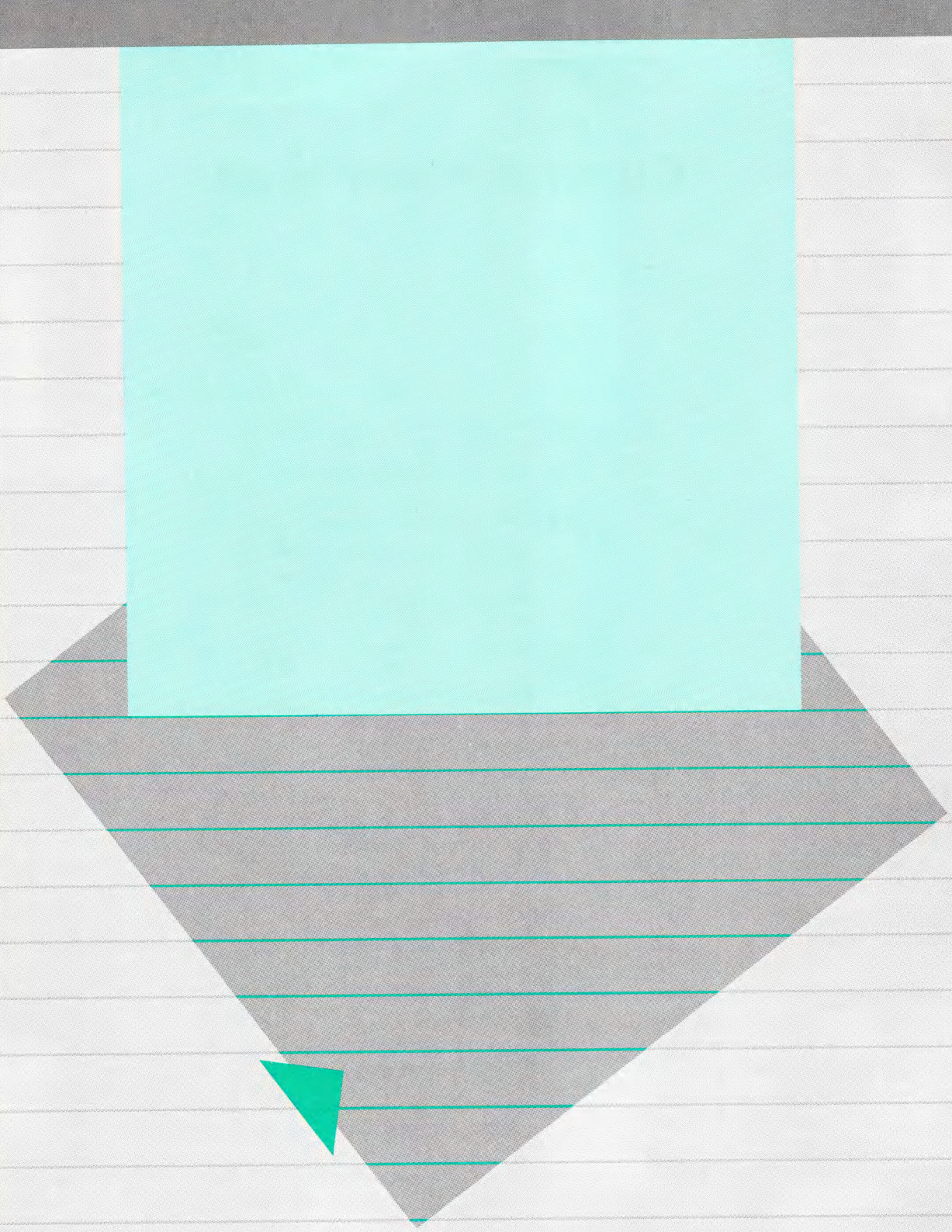
```
Screen boxes represent information as it will
appear on the computer's display screen.
```

Throughout the manual, extensive use has been made of marginal notes for key points, definitions, and cross-references. After reading a chapter or section, you can use the marginal notes to review what you've learned or to refer back to a particular point for quick reference.

New terms being introduced for the first time are set in *italics;* definitions for most such terms can be found in the marginal notes, the Glossary, or both.

Numbers (such as memory addresses) preceded by a dollar sign, such as $9600, are expressed in hexadecimal; numbers without a dollar sign are generally in decimal, unless otherwise stated.

General Information



General Information

BASIC: Beginner's All-purpose Symbolic Instruction Code

ANSI: American National Standards Institute

Applesoft BASIC is a very extended version (in computer parlance, a *superset*) of the BASIC programming language. It includes many more features than either the original BASIC, developed at Dartmouth College in the 1960s, or the standard version of the language, as defined by the American National Standards Institute (ANSI). The extra features allow your programs to use the special capabilities of the Apple IIe, such as color graphics, animation, and hand controls.

This first chapter introduces the Applesoft language and the environment in which it operates. Here you will find information on how to create, modify, execute, and store Applesoft programs.

creating and modifying programs: see Section 1.1

Section 1.1, "Statements and Lines," deals with the fundamental units of Applesoft programs. It tells how to type Applesoft statements for immediate execution and how to create and modify programs in the computer's memory.

operations on whole programs: see Section 1.2

Section 1.2, "Operations on Whole Programs," introduces Applesoft's commands for displaying a program on the screen, writing it to an output device such as a printer, executing it, saving it on a disk, and retrieving it from a disk.

interrupting and resuming: see Section 1.3

Section 1.3, "Interrupting and Resuming a Program," tells how to suspend or cancel the execution of a running program and how to resume execution after an interruption.

on-screen editing: see Section 1.4

Section 1.4, "Editing What You Type," briefly describes Applesoft's facilities for correcting typing errors and editing text on the screen.

Statements and Lines

1.1

program line: the basic unit of an Applesoft program

statement: a unit of a program specifying an action for the computer to perform

The basic unit of an Applesoft program is the *program line*, which may contain one or more *statements* specifying actions you want the computer to perform. Most Applesoft statements are identified by

one or more *keywords*, special words that Applesoft recognizes as denoting a particular type of statement.

You can type a program line whenever you see Applesoft's *prompt character*, a right bracket (]), displayed on the screen followed by the cursor. Each line you type must end in a press of the RETURN key (but see Section 1.1.4 about multiple statements per line). Depending on what you type, the statements in the line may either be executed immediately or *deferred* for later execution as part of a complete program.

deferred execution: see Section 1.1.2

Use CAPS LOCK while typing Applesoft programs

Applesoft understands only uppercase letters. Most programmers therefore keep the CAPS LOCK key down while typing programs.

Notice that a program line is not the same thing as a line of text on the screen. If the cursor reaches the end of a screen line while you're typing a program line, it will "wrap around" to the beginning of the next screen line and continue displaying what you type. Although the screen is only 40 columns wide (or 80 if you're using the Apple IIe 80-Column Text Card), a program line may be up to 239 characters long and ends only when you press the RETURN key.

80-Column Text Card: see *Apple IIe Owner's Manual, Apple IIe 80-Column Text Card Manual*

Program lines may be up to 239 characters long

Actually, you can type as many as 255 characters in a program line, but all characters after 239 will be ignored. If you type more than 255 characters, Applesoft will display a backslash character (\) and cancel the entire line. It will then redisplay the prompt character (]) followed by the cursor, and you will have to retype the entire line from the beginning. As a warning, Applesoft will "beep" the computer's built-in speaker with every character you type beginning with the 245th in a line.

It's usually a bad idea to type program lines this long. In practice, you should keep your lines well below 239 characters in length.

1.1.1 *Immediate Execution*

If you want Applesoft to execute a program line as soon as you type it, just type the line and press the RETURN key. For example, if you type

PRINT **statement:** see Section 5.2.2

```
PRINT "HELLO"
```

Applesoft immediately displays the word HELLO on the screen, on the line following what you just typed.

1.1.2 *Line Numbers and Deferred Execution*

line number: a number identifying a line in an Applesoft program

If you want Applesoft to save a program line to be executed later—that is, if you want it to *defer* execution—then precede the line with a *line number:*

```
10 PRINT "HELLO"          —10 is the line number
```

Maximum line number is 63999

Line numbers must be in the range 0 through 63999. Applesoft uses the presence or absence of line numbers to determine whether the line you type is to be carried out immediately or deferred (stored for execution at some future time).

program: a sequence of program lines, each with a different line number

A sequence of deferred-execution lines, each preceded by a different line number, is an Applesoft *program*. Program lines are stored in the computer's memory in sequential order, from the lowest-numbered line to the highest.

1.1.3 *Adding Lines to a Program*

To add a new line to a program, just type the new line preceded by a line number indicating where in the program you wish to insert it. It makes no difference in what order you enter program lines; Applesoft will put them in the proper sequential order for you.

Program lines automatically sorted into proper sequential order

Leave intervals between line numbers

Helpful Hint: Instead of using consecutive line numbers (0, 1, 2, . . .), it's usually more convenient to leave intervals of 5 or 10 or 20 between the line numbers in your program. This makes it easy to insert new lines, if necessary, in between the old ones.

1.1.4 *Multiple Statements on the Same Line*

Colons separate multiple statements

Applesoft allows you to put more than one statement on the same program line. Use a colon (:) to separate the statements:

```
40 PRINT "COME OUTSIDE" : PRINT "AND
     PLAY"
```

You can type as many statements as will fit within the limit of 239 characters per line.

Multiple statements make editing more difficult (although they speed up program execution)

Although using multiple statements on the same line can speed up the execution of your program, it can also make program editing difficult and time-consuming. The example above, for instance, has two statements on the same line. In order to change the word OUTSIDE in the first statement to INSIDE, you would have to retype both statements. But if each statement were on its own line, you would have to retype only the one statement you want to change. This may not seem like much of a time saving; but when you multiply three or four seconds by the hundreds of edits you might need to make in developing a typical program, the savings can become considerable.

1.1.5 *Deleting Lines from a Program: The DEL Command*

```
DEL 100, 200
```

DEL deletes lines from the program in memory

The DEL command deletes (removes) a range of consecutive lines from the program currently in memory. The line numbers of the first and last lines to be deleted follow the keyword DEL and are separated from each other by a comma. All program lines between the two specified line numbers, inclusive, are deleted from the program. The example above, for instance, will delete all lines from 100 to 200, inclusive.

If either line number is out of the range of lines in the actual program (for instance, if the command is DEL 100, 200 and the highest existing line number is 150), then all existing lines within the specified range are deleted. If DEL specifies a range of lines that doesn't exist, or if the second line number is smaller than the first, the command has no effect:

```
DEL 200, 100
```
—nothing happens

A single number with a comma also has no effect:

```
DEL 35,
```
—nada pasa

A single number without a comma is a syntax error:

```
DEL 35
```
—syntax error

Deleting a single line

To delete a single line from the program, simply type the number of that line and press RETURN:

```
150
```
—press the RETURN key after you type the line number

If you're fond of redundancy, you can also use DEL 150,150 to do the same thing.

Dash not allowed in DEL command

Unlike the LIST command, you cannot use a dash (-) to separate the line numbers in the DEL command:

```
DEL 60 - 100          —causes a syntax error
```

LIST **command:** see Section 1.2.3

The DEL command is normally used in immediate execution. You can also use it from within a program, but as soon as it is executed the program will stop with no error message:

```
20 DEL 135, 250       —lines 135 through 250 removed from program; program execution halts
```

1.1.6 *Changing Lines in a Program*

To alter or replace an existing line of your program, simply type the new line using the same line number as the existing one. What you type will replace the old line under the same line number; the old line will be forgotten.

1.1.7 *Annotating a Program: The REM Statement*

```
REM TEST FOR ERROR
```

REM **is for including explanatory remarks to a human reader**

One rule of good programming practice is to include comments in your program, explaining or clarifying to a human reader how the program works. Applesoft's REM statement allows you to include such remarks within the body of your program. It consists of the keyword REM (for "remark") followed by any explanatory notes you care to include. For example,

```
0 REM MONTHLY BUDGET PROGRAM
```

LIST **command:** see Section 1.2.3

RUN **command:** see Section 1.2.4

This statement is included in the program strictly for the benefit of the human reader. When you list the program, the REM statement will appear just like any other statement. But when you run the program, Applesoft will ignore the REM statement and just go on to the next line. Everything following the keyword REM on the same line will be ignored. See Chapter 8, "Bringing It All Together," for some tips on the use of the REM statement.

1.2 Operations on Whole Programs

This section describes Applesoft's commands for manipulating whole programs:

NEW **command:** see Section 1.2.1

- NEW clears the current program from the computer's memory so you can start typing another.

CLEAR **command:** see Section 1.2.2

- CLEAR resets all variables and internal control information to their initial settings without affecting the Applesoft program in memory.

LIST **command:** see Section 1.2.3

- LIST displays the current program on the screen or writes it to an output device such as a printer.

RUN **command:** see Section 1.2.4

- RUN executes the program currently in memory. It can also be used to load and execute a program stored on a disk.

SAVE **command:** see Section 1.2.5

- SAVE writes the program currently in memory onto a disk or a tape cassette for future use.

LOAD **command:** see Section 1.2.6

- LOAD reads a program into memory from a disk or a tape cassette for execution.

You can use all of these commands for immediate execution; you can use some of them from within your Applesoft programs as well.

1.2.1 *The* NEW *Command*

```
NEW
```

NEW clears memory for a new program

The NEW command clears the current program from memory, resets the values of all numeric variables to 0 and those of all string variables to the null string, and prepares Applesoft to accept a new program. If there are no program and no variables in memory, NEW has no effect.

variables: see Section 2.1

NEW in deferred execution

Although NEW is usually used in immediate execution, you can also use it in deferred execution (from within a program):

```
100 IF A$ = "RATS" THEN NEW
                          —NEW in conditional statement
999 NEW                   —NEW on its own line
```

Warning

Using NEW in deferred execution can do strange and unpredictable things to Applesoft's innards, causing subsequently entered programs to hang. If you use NEW from within a program, it's a good idea to warn your user to restart the system before typing another program:

hang: for a program to "spin its wheels" indefinitely, performing no useful work

```
100 IF A$ = "RATS" THEN PRINT
      "PLEASE RESTART YOUR SYSTEM
      BEFORE TYPING A NEW PROGRAM.": NEW
```

1.2.2 *The* CLEAR *Command*

```
CLEAR
```

The CLEAR command resets the values of all numeric variables to 0 and those of all string variables to the null string; it also resets Applesoft's internal control information to its initial state. It has no effect on the program lines in memory.

null string: a string containing no characters

CLEAR in deferred execution

Although CLEAR is usually used in immediate execution, you can also use it in deferred execution (from within a program):

```
100 IF Z$ = "NUTS" THEN CLEAR
                          —CLEAR in conditional
                           statement
999 CLEAR                 —CLEAR on its own line
```

Warning

Be careful where you execute CLEAR. Since CLEAR resets Applesoft's internal control stack, using it in the midst of a subroutine or in a FOR/NEXT loop can interfere with the orderly flow of program execution. The following program, for example, will fail in line 30 with a NEXT WITHOUT FOR error:

```
10 FOR X = 1 TO 10        —try to loop 10 times
20 PRINT X
30 CLEAR                  —CLEAR resets control stack
                           (among other things)
40 NEXT X                 —program fails here—doesn't
                           know it's in a loop
50 PRINT "HI!"            —program won't get this far
```

subroutines, control stack: see Section 3.4

FOR/NEXT loops: see Section 3.3

1.2.3 *The* LIST *Command*

```
LIST
LIST 100
LIST 100,
LIST - 200
LIST ,200
LIST 100, 200
LIST 100 - 200
```

LIST displays or prints a program

The LIST command displays on the screen all or part of the program currently in memory, or writes it to the current output device as specified in the last PR# statement. (For example, if there is a printer connected to slot 1, and if the statement PR# 1 has been executed, then the program listing is sent to the printer.)

PR# statement: see Section 5.2.1

Listing the entire program

To list the entire program, just type the keyword LIST and press RETURN:

```
LIST
```

Listing a portion of the program

You can list a portion of the program by specifying the first and last lines you want to list, separated by either a comma or a dash:

```
LIST 100, 250             —display lines 100 through
                           250
LIST 100 - 250            —also display lines 100
                           through 250
```

If none of the lines in the specified range are in memory, nothing will be listed; if the specified range is greater than the actual range of lines in the program, Applesoft will list the entire program.

If you specify only one line number preceded by a comma or dash, all lines from the beginning of the program through the specified line will be listed:

```
LIST ,100
```
—display from beginning of program through line 100

If you specify only one line number followed by a comma or dash, all lines from the specified line through the end of the program will be listed:

```
LIST 100 -
```
—display from line 100 through end of program

If you just specify a single line number, only that line will be listed:

```
LIST 100
```
—display line 100 only

You cannot list line number 0 by itself. You'll have to use a form like

```
LIST ,1
```

Warning

Always be sure to type the keyword LIST before the number of the program line you want to list; typing a line number not preceded by a keyword deletes the specified line from the program (see Section 1.1.5, "Deleting Lines from a Program: The DEL Command").

LIST in deferred execution

Although the LIST command is usually used in immediate execution, you can also use it from within a program:

```
150 LIST
```
—list entire program

```
235 IF Z = X THEN LIST 10,75
```
—list lines 10 through 75 if variable Z holds same value as variable X

LIST statements within a program can be particularly useful in debugging. With them, you can test for various error conditions and display or print only the section of the program in which the error occurred.

1.2.4 *The* RUN *Command*

```
RUN
RUN 275
RUN MONTHLY BUDGET
```

RUN executes a program

The RUN command instructs Applesoft to execute the program currently in memory. If no line number is given, execution begins at the beginning of the program; if the RUN command includes a line number, execution begins at the specified line:

```
RUN                    —execute program from beginning
RUN 500                —execute program from line 500
```

If you attempt to run a program from a specified line number (as in RUN 500) and that line doesn't exist, the message

```
?UNDEF'D STATEMENT ERROR
```

will be displayed and program execution will halt.

RUN in deferred execution

Although RUN is normally used in immediate execution, you can also use it from within a program:

```
150 IF A = 0 THEN RUN
                       —if value of A is 0, then execute program from beginning
235 RUN 600            —execute program from line 600
```

You can use this technique, for example, to restart a game or to avoid executing some code with low line numbers.

Warning

Whenever the RUN statement is executed, it resets the values of all numeric variables to 0 and those of all string variables to the null string before executing the first program line. If you have assigned values to any variables in immediate execution, those values will be forgotten. This happens even if there is no program currently in memory.

variables: see Section 2.1

Running a program from a disk

If your computer is equipped with a disk drive and the Disk Operating System (DOS) is active, you can use the RUN command to load a program into memory from a disk file and execute it. To do this, follow the keyword RUN with the file name under which the program is

stored on the disk. For example, if the program you want to run is stored in a file named AWAY, first make sure the disk containing that file is in the disk drive, then type

```
RUN AWAY
```

and press RETURN. Applesoft (and DOS) will do the rest.

If you try to use this form of the RUN command with no disk drive connected to your computer, or without DOS loaded and active, you'll get a syntax error.

For more information on disk drives, disks, files, and file names, see the DOS manual that came with your disk drive. For related Applesoft commands, see Sections 1.2.5, "The SAVE Command," and 1.2.6, "The LOAD Command." For information on using a cassette tape recorder in place of a disk drive, see Appendix M, "If You Have a Cassette Recorder."

1.2.5 *The SAVE Command*

```
SAVE
SAVE MONTHLY BUDGET
```

SAVE writes a program to a disk or tape

On systems equipped with a disk drive, the SAVE command writes the Applesoft program currently in memory to a file on a disk. The keyword SAVE is followed by the file name under which the program is to be written. The copy of the program in memory is not affected in any way. For example,

```
SAVE MY CHILD          —store current program on disk
                          under file name MY CHILD
```

Attempting to use this form of the SAVE command with no disk drive connected to your computer, or without the Disk Operating System (DOS) loaded and active, will result in a syntax error.

Saving programs on tape: see Appendix M

CONTROL - RESET : see Section 1.3.2

If you issue the SAVE command without specifying a file name, Applesoft will attempt to write the program in memory onto a tape cassette. If no cassette recorder is connected, the computer will seem to hang for a while; the actual time that will pass before you regain control depends on the length of the program in memory. You can regain control immediately by pressing CONTROL - RESET .

For more information on disk drives, disks, files, and file names, see the DOS manual that came with your disk drive. For related Applesoft commands, see Sections 1.2.4, "The RUN Command," and 1.2.6, "The LOAD Command." For information on using a cassette tape recorder in place of a disk drive, see Appendix M, "If You Have a Cassette Recorder."

1.2.6 *The* LOAD *Command*

```
LOAD
LOAD MONTHLY BUDGET
```

LOAD reads a program from a disk or tape

On systems equipped with a disk drive, the LOAD command reads an Applesoft program from a file on a disk into the computer's memory for execution or editing. The keyword LOAD is followed by the file name under which the program is to be found on the disk. For example,

```
LOAD THE DICE          —load program into memory
                          from file named THE DICE
```

LOAD does not execute the program it retrieves; it merely reads a copy of the program into memory. You can then execute the program, if you wish, with the RUN command. The copy of the program on the disk is not affected in any way.

RUN **command:** see Section 1.2.4

If the disk in the disk drive doesn't contain a file of the specified name, the error message

```
FILE NOT FOUND
```

will be displayed. If there is no disk drive connected to your computer, or if the Disk Operating System (DOS) isn't loaded and active, you'll get a syntax error.

Loading programs from tape: see Appendix M

CONTROL-RESET: see Section 1.3.2

If you issue the LOAD command without specifying a file name, Applesoft will attempt to read a program into memory from a tape cassette. If no cassette recorder is connected, or if the tape in the recorder doesn't contain a program to load, or if the recorder is turned off, the computer will hang forever looking for a program that isn't there. When you get bored waiting, press CONTROL-RESET to regain control.

For more information on disk drives, disks, files, and file names, see the DOS manual that came with your disk drive. For related Applesoft commands, see Sections 1.2.4, "The RUN Command," and 1.2.5, "The SAVE Command." For information on using a cassette tape recorder in place of a disk drive, see Appendix M, "If You Have a Cassette Recorder."

1.3 Interrupting and Resuming a Program

If a program starts to run away from you, there are various ways of interrupting it and regaining control. This section covers Applesoft's facilities for getting out of problem programming situations, infinite loops, and the like.

1.3.1 Suspending Screen Output

CONTROL-S temporarily suspends screen output

Quite often the output a program displays, or the listing of the program itself, exceeds the number of lines available on the display screen, causing the output to fly by on the screen too fast for you to read. In such cases, you can press CONTROL-S (type the letter S while holding down the CONTROL key) to suspend the output of text to the screen temporarily so that you can comfortably read what's there. CONTROL-S doesn't permanently discontinue the display of text; pressing any key, including another CONTROL-S, causes screen output to resume. You can then suspend it again with another CONTROL-S. To discontinue a program or a listing permanently, use CONTROL-C.

CONTROL-C: see Section 1.3.2

Helpful Hint: Experienced programmers looking at listings of long programs keep the CONTROL key continually pressed; they control the listing by pressing the S key whenever they want to suspend or continue it.

1.3.2 Interrupting Program Execution

Applesoft gives you two ways of interrupting the execution of a running program or canceling a listing. Pressing CONTROL-C interrupts the program in such a way that it is usually possible to resume execution from the point of the interruption; CONTROL-RESET is somewhat more drastic, and often leaves the system in a state from which the program can't be resumed with a CONT statement.

CONTROL-C

CONTROL-C cancels execution or listing of a program

Pressing CONTROL-C (typing the letter C while holding down the CONTROL key) cancels the execution or listing of a program and returns Applesoft to its command level, displaying the prompt character (]). You can then resume execution of the program, if you wish, with the CONT command. To cancel execution of a program that is waiting for a response to an INPUT statement, CONTROL-C must be the first character typed and must be followed immediately by RETURN.

INPUT statement: see Section 5.1.2

GET statement: see Section 5.1.3

Interrupting a GET: CONTROL-C will not interrupt a program waiting for a response to a GET statement; unlike the INPUT statement, GET will assume that CONTROL-C is a valid response and will assign the ASCII code for the character CONTROL-C to the specified variable. To allow a program halted at a GET statement to be interrupted with CONTROL-C, use this form:

```
250 GET A$                          —wait for user to press a key
260 IF A$ = CHR$(3) THEN STOP
                                    —if user presses CONTROL-C
                                      (ASCII code 3), then stop
```

ASCII code: see Section 4.2.1 and Appendix C

Warning

In certain situations, using CONTROL-C can disconnect the disk operating system. See the DOS manual for information on this point.

CONTROL-RESET

CONTROL-RESET unconditionally stops any program or command

In most cases you can immediately and unconditionally stop the execution of any Applesoft program or command by pressing CONTROL-RESET (pressing the RESET key while holding down the CONTROL key). The program in memory remains intact, but some of Applesoft's internal "housekeeping" information is changed; as a result, it may not be possible to resume execution of the program with the CONT command.

Controlling CONTROL-RESET: Your Apple IIe has an advanced software feature called a *reset vector,* which allows you to control what happens when CONTROL-RESET is pressed. You can use the reset vector to make the program continue as if nothing had happened, branch to some other portion of the program, or do whatever you choose. Use of this technique requires knowledge of the Apple IIe's built-in Monitor program: see the *Apple IIe Reference Manual* for details.

Apple IIe Monitor program: see *Apple IIe Reference Manual*

1.3.3 *Resuming Program Execution: The CONT Command*

```
CONT
```

CONT continues execution after an interruption

STOP statement: see Section 3.6.1

END statement: see Section 3.6.2

CONTROL-C: see Section 1.3.2

The CONT (for "continue") command is used to resume execution of a program after it has been interrupted by a STOP or END statement or by pressing CONTROL-C. Execution will continue at the first statement after the STOP or END, or at the point in the program where execution was interrupted by CONTROL-C.

When CONT won't work

CONT won't work if

- the program has been stopped because of an error
- an error has occurred in immediate execution
- an INPUT statement has been interrupted with CONTROL-C
- any program line has been edited since the program stopped running

INPUT statement: see Section 5.1.2

However, you can continue the program with CONT after examining or changing the values of variables, provided you haven't edited any program lines.

When a program is interrupted with CONTROL-RESET, CONT may or may not be able to continue execution. Let the programmer beware!

Warning
The CONT command should be used in immediate execution only. If it is executed from within a program, it will cause the program to hang.

1.4 *Editing What You Type*

Tutorials abound...

This section gives a very brief description of Applesoft's facilities for correcting typing mistakes and editing text on the screen. More detailed discussions of these features can be found in the *Apple IIe Owner's Manual* and the Apple IIe *Applesoft Tutorial.* For hands-on experience with the various keys and editing features, use the APPLE PRESENTS ... APPLE training disk.

1.4.1 Canceling an Input Line

CONTROL-X cancels a line of input

CONTROL-X is your "escape hatch." By typing the letter X while holding down the CONTROL key, you can change your mind (as long as you haven't yet pressed the RETURN key) and cancel a program line that you're entering or editing or a line of input that you're in the midst of typing to a program. Applesoft will display a backslash (\) at the end of the line you were typing, to show that it's ignoring that input, and will redisplay the cursor at the beginning of the next line of the screen.

- If you were typing a new program line, the whole line is eliminated and you can start over again.
- If you were retyping a previously entered program line, any changes you had typed will be canceled.
- If you were typing input to a running program, the line you were typing is ignored and the program waits for your new response.

CONTROL-X does not affect any previous input you've typed or program lines already entered.

1.4.2 The Arrow Keys

There are four arrow keys on the Apple IIe keyboard:

- The LEFT-ARROW key works as a backspace. It moves the cursor one position to the left and "erases" the last character typed from the keyboard (or recopied with the RIGHT-ARROW key; see below). No characters are removed from the screen, but the last character typed is forgotten, as if it had never been typed.
- The RIGHT-ARROW key "recopies" the character under the cursor as if it had been typed from the keyboard, then moves the cursor one position to the right. Moving the cursor over a character with RIGHT-ARROW is exactly the same as typing that character from the keyboard.
- The DOWN-ARROW moves the cursor down one line without erasing or recopying any characters.
- The UP-ARROW key has no effect in Applesoft.

Notice that the LEFT-ARROW key doesn't erase any characters from the screen; it just tells Applesoft to "forget" the last character it received. If any pure cursor moves have been used, the character "erased" may not even be the one the cursor backs up over.

pure cursor moves: see Section 1.4.3

escape mode: see Section 1.4.3

In escape mode, all four arrow keys function as pure cursor moves, equivalent to I (up), J (left), K (right), and M (down). That is, they lose their backspace and recopy functions and simply move the cursor one position in the indicated direction, remaining in escape mode. To cancel escape mode after moving the cursor, press the SPACE bar.

The Apple IIe keyboard's auto-repeat feature is particularly handy for long cursor moves. If you press and hold down any of the arrow keys, the cursor will move repeatedly in the indicated direction for as long as you hold down the key. (Exception: the UP-ARROW key doesn't move the cursor unless you're in escape mode.)

INPUT **statement:** see Section 5.1.2

GET **statement:** see Section 5.1.3

ASCII: see Section 4.2.1 and Appendix C

For Experts Only: The UP-ARROW and DOWN-ARROW keys can be typed by the user in response to an INPUT statement in a running Applesoft program. (LEFT-ARROW and RIGHT-ARROW can't be, because they're interpreted as backspace and recopy, even in program input; but any of the four arrow keys can be typed as a response to a GET statement.) In your own programs, you can make the arrow keys mean just what you choose them to mean (neither more nor less) by having the program test the input for each arrow's ASCII value, as shown in Table 1-1. The program can then take any action you want on receiving one of these codes from the user.

("The question is," said Alice, "whether you *can* make keys mean so *many* different things."

"The question is," said Humpty Dumpty, "which is to be master—that's all.")

Table 1-1 ASCII Equivalents of Arrow Keys

Key	ASCII Code	Keyboard Equivalent
LEFT-ARROW	8	CONTROL-H
RIGHT-ARROW	21	CONTROL-U
UP-ARROW	11	CONTROL-K
DOWN-ARROW	10	CONTROL-J

1.4.3 *Escape Mode*

ESC alters meanings of some keys

Pressing the ESC (for "escape") key puts Applesoft into a state called *escape mode*, in which certain keys take on special meanings. Some of the keys become *pure cursor moves,* meaning that they move the cursor around on the screen without erasing or recopying characters or affecting Applesoft's input in any way. Others can be used to clear away all text from all or part of the screen, again without having any effect on the input received by Applesoft.

Although Applesoft normally doesn't understand lowercase letters, it will accept them in escape mode. All of the letter keys listed below will have the same effect whether they are typed in upper- or lowercase.

In escape mode, the following characters move the cursor one position in the stated direction and then leave escape mode. To continue moving the cursor, you have to press the ESC key again. The functions of these keys are illustrated in Figure 1-1.

Figure 1-1 Single Cursor Moves

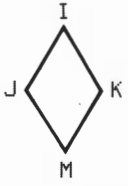
A, B, C, D move cursor one position

- A moves the cursor one position to the right.
- B moves the cursor one position to the left.
- C moves the cursor down one line.
- D moves the cursor up one line.

The following characters move the cursor one position in the stated direction and remain in escape mode. You can then continue moving the cursor without pressing the ESC key again. These keys are especially useful for long-range cursor moves. The functions of these keys are illustrated in Figure 1-2.

Figure 1-2 Long-range Cursor Moves

I, J, K, M are for long-range moves



- I moves the cursor up one line.
- J moves the cursor one position to the left.
- K moves the cursor one position to the right.
- M moves the cursor down one line.

Notice that the I, J, K, and M keys form a diamond shape on the keyboard, representing the directions in which these keys move the cursor (I up, J left, K right, M down).

Arrows also work for long-range moves

The four arrow keys function in escape mode exactly the same as I, J, K, and M. That is, they move the cursor one position in the indicated direction and remain in escape mode.

The Apple IIe keyboard's auto-repeat feature is particularly handy for long cursor moves. If you press and hold down I, J, K, M, or any of the arrow keys while in escape mode, the cursor will move repeatedly in the indicated direction for as long as you hold down the key.

In escape mode, the following keys clear away all text from all or part of the display screen and then leave escape mode:

E, F, @ clear all or part of the screen

- E clears from the current cursor position to the end of the line.
- F clears from the current cursor position to the end of the text window.

text window: see Section 5.2.4

- @ clears the entire text window and moves the cursor to the top-left corner.

The special functions of all keys in escape mode are summarized in Table 1-2. To leave escape mode, press any key except one of those listed in the table.

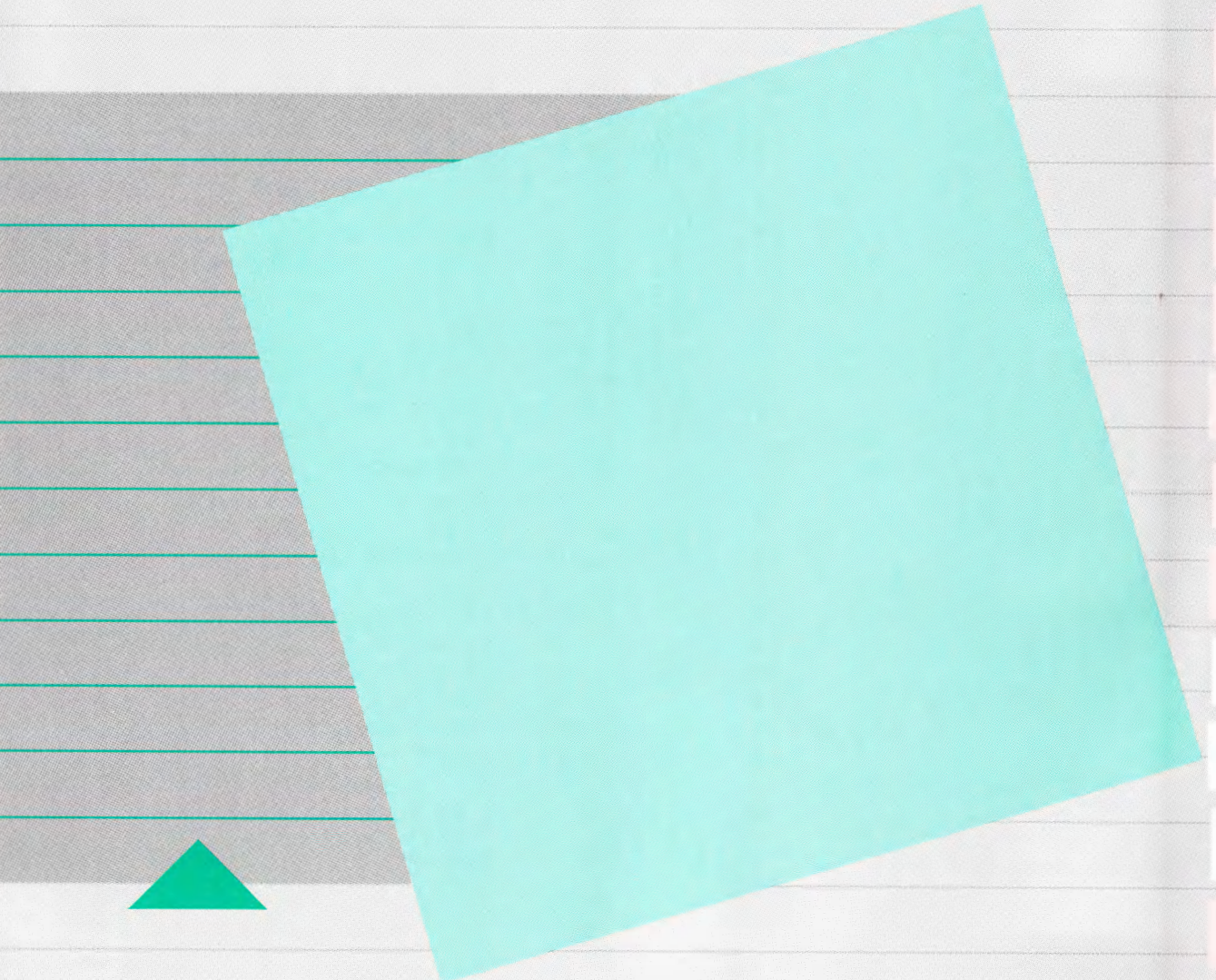
Leaving escape mode

To avoid inadvertently pressing a key that has a special meaning, it's safest always to use the SPACE bar to leave escape mode.

Table 1-2 Escape-Mode Functions

Key	Function
A	Moves cursor right one position; leaves escape mode
B	Moves cursor left one position; leaves escape mode
C	Moves cursor down one line; leaves escape mode
D	Moves cursor up one line; leaves escape mode
I	Moves cursor up one line; remains in escape mode
J	Moves cursor left one position; remains in escape mode
K	Moves cursor right one position; remains in escape mode
M	Moves cursor down one line; remains in escape mode
LEFT-ARROW	Moves cursor left one position; remains in escape mode
RIGHT-ARROW	Moves cursor right one position; remains in escape mode
UP-ARROW	Moves cursor up one line; remains in escape mode
DOWN-ARROW	Moves cursor down one line; remains in escape mode
E	Clears from cursor to end of line; leaves escape mode
F	Clears from cursor to end of text window; leaves escape mode
@	Clears entire text window; moves cursor to top-left corner; leaves escape mode

Variables and Arithmetic



Variables and Arithmetic

This chapter deals with variables and arithmetic in Applesoft. These concepts are fundamental to Applesoft programming and will appear again and again throughout this manual.

variables: see Section 2.1

Section 2.1, "Variables," discusses how to define and use variables, the various types of variable available in Applesoft, and the rules for naming them.

assignment statement: see Section 2.2

Section 2.2, "Assigning Values to Variables: The Assignment Statement," deals with one of Applesoft's most basic types of statement, the assignment statement.

expressions, precedence rules: see Section 2.3

Section 2.3, "Expressions," discusses arithmetic operators and expressions and the rules of precedence that govern them.

functions: see Section 2.4

Section 2.4, "Functions," covers Applesoft's built-in arithmetic functions and tells how you can define your own functions.

Variables

2.1

variable: a symbol representing a location in the computer's memory where a value can be stored

A *variable* is a symbol representing a location in the computer's memory where a value can be stored. The first time your program *assigns* a value to a particular variable, Applesoft automatically allocates a memory location or locations for that variable and stores the specified value at that location. Thereafter, whenever your program uses that particular variable name, Applesoft will take the name to refer to the value stored at the corresponding location. For instance, if the variable `PI` refers to a memory location where the value `3.14159` is stored, then the statement `PRINT PI` will display the value `3.14159` on the screen.

Variable types

real variables: see Section 2.1.2

Applesoft has three types of variable:

- *Real* variables can contain either whole numbers or numbers containing decimal fractions.

integer variables: see Section 2.1.3

string variables: see Section 2.1.4

arrays: see Section 2.1.5

- *Integer* variables can contain whole numbers only.
- *String* variables can contain strings of text characters such as words or names.

In addition, Applesoft allows you to define collections of variables, called *arrays,* of any of the types listed above.

Reals save time; integers save space

Programming Tip: Applesoft converts all integer values to real form before performing arithmetic on them. Because this conversion takes time, integer arithmetic is considerably slower than arithmetic on real quantities. However, integers take up less space in the computer's memory than real numbers. In relatively small programs in which space is not a concern, you can speed up your program by using real variables instead of integers wherever possible, especially in subroutines, loops, and other sections of code that are executed many times. In large programs where space is critical, you can save space at the expense of time by using integers instead of reals, particularly in arrays containing many elements. See Appendix G, "Hints for Program Efficiency," for further suggestions on how to save space and time in your programs.

subroutines: see Section 3.4

loops: see Section 3.3

2.1.1 *Variable Names*

Rules for variable names

The name of a variable must begin with a letter of the alphabet, which may be followed by one or more letters and/or digits. In addition, the names of all integer variables must end with a percent character (%) and those of string variables must end with a dollar sign ($). The various variable types and the rules for naming them are summarized in Table 2-1.

Table 2-1 Variable Types

Type	Symbol	Simple Examples	Array Examples
Real	(none)	K	AGE (CHILD)
		PRICE	TAX (ITEM)
		N1	N1 (J%, 3)
Integer	%	J%	YEAR% (N)
		G5%	BOOK% (COUNT)
		N1%	N1% (J%, 3)
String	$	A$	SHOP$ (5)
		SAM$	DAY$ (WEEK)
		N1$	N1$ (J%, 3)

A variable name can be up to 239 characters long, but Applesoft uses only the first two characters to distinguish one variable from another of the same type. All characters beyond the first two in a name are ignored, so long as they don't include a reserved word (see below).

Don't begin variable names of the same type with the same first two characters

Take care not to begin the names of different variables of the same type with the same two characters. Applesoft will consider the names SUM and SUNSTROKE, for example, to refer to the same variable, since they both begin with the same two characters.

Notice that the restriction above applies only to variables of the same type. The names TAX, TAX%, and TAX$ refer to three different variables, even though they all begin with the same two characters, because they are of different types (real, integer, and string). However, names of the same type that begin with the same two characters—such as TAX and TAXABLE, THIS% and THIN% or OTTER$ and OTHER$—refer to the same variable.

Reserved words illegal in variable names

Reserved Words: Certain words used in Applesoft are *reserved* for special uses in specific commands; you can't use these words as variable names or as parts of variable names (even beyond the first two characters). For instance, TOTAL or SUBTOTAL would be illegal as variable names, because they both contain the reserved word TO. See Appendix D, "Reserved Words," for a list of Applesoft's reserved words.

2.1.2 *Real Variables*

Range of real values

A real variable can hold any numeric value, with or without a decimal point, between -9.99999999E+37 and +9.99999999E+37 (where "E+37" means "times 10 to the +37th power"). Applesoft represents real numbers to 32 bits (about 9 digits) of precision.

Real variable names consist of letters and digits only

The name of a real variable must consist of letters and digits only. Some legal real variable names are

```
SAM
TAX
Q7
SUMOFALLNUMBERS
```

Real variables preset to 0

Until they are given some other value with an assignment statement, all real variables are preset to the value 0.

2.1.3 *Integer Variables*

Integer variables can hold only whole-number values between −32767 and +32767. The name of an integer variable must

Integer variable names end with %

end with the percent character (%). Some legal integer variable names are

```
SHARE%
D5%
TAX%
```

Integer variables preset to 0

Until they are given some other value with an assignment statement, all integer variables are preset to the value 0.

Real values assigned to integer variables are **truncated, not rounded**

If a number containing a decimal fraction is assigned as the value of an integer variable, it is *truncated* to the next lowest whole number—*not* rounded to the nearest whole number:

```
LET A% = 32.678
```
—value 32 assigned to variable A%

```
LET B% = -34.2
```
—value −35 assigned to variable B%

2.1.4 *String Variables*

string: a sequence of text characters; see Section 4.2

A *string* is a sequence of text characters (letters, digits, and punctuation marks). Just as you can write numeric constants such as 27 and 2.236 in your Applesoft programs, you can write *string constants* by enclosing the characters in the desired string between double quotation marks:

String constants enclosed in double quotation marks

```
"NOT WITH A BANG BUT A WHIMPER"
"George Bernard Shaw"
"H234J7"
"$?*!!%"
```

Even though Applesoft doesn't understand lowercase letters when you use them in keywords, it will allow you to use them in a string constant, as the second example above shows.

null string: a string containing no characters

A string can contain from 0 to 255 characters; when it contains no characters it is called a *null string*. Two quotation marks with nothing between them denote the null string:

```
""
```
—a string with no characters

String variable names end with $

A string variable can hold any string as its value. Its name must end with a dollar sign ($). Some legal string variable names are

```
NAME$
S9$
J$
```

String variables preset to null string

Until they are given some other value with an assignment statement, all string variables are preset to the null string.

2.1.5 *Arrays: Collections of Variables*

array: a collection of variables referred to by the same name and distinguished by means of numeric subscripts

An *array* is a collection of variables referred to by the same name, usually holding a collection of data items that are related to each other in some logical or systematic way. The individual variables in the array are called its *elements,* and are distinguished from one another by means of identifying index numbers called *subscripts.*

simple variable: a variable that is not an element of an array

An array can be of any type: integer, real, or string. Array names follow the same rules as simple variable names of the same type. To refer to a particular element of an array, write the array name followed by one or more subscripts, separated by commas and enclosed in parentheses. The subscripts refer to the position of the desired element within the array:

`Q (6)`	—element 6 of real array Q
`FIGURE% (N)`	—element N of integer array FIGURE%
`NAME$ (J - 3)`	—element J - 3 of string array NAME$
`COUNT (SUM%, 2)`	—element (SUM%, 2) of real array COUNT

Figure 2-1 A Typical Array

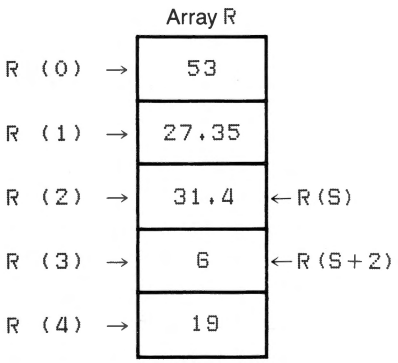


Figure 2-1 shows a real array named R with five elements, numbered 0 to 4. Element R(0) (pronounced "R-sub-zero") holds the value 53, R(1) holds 27.35, and so on. If the value of variable S is 2, then the expression R(S) refers to element R(2), whose value is 31.4, and the expression R(S + 2) refers to element R(4), which holds the value 19.

For a fuller discussion of arrays and their use, see Section 4.1, "Arrays."

2.2 Assigning Values to Variables: The Assignment Statement

```
LET PI = 3.14159265
COUNT% = 0
```

Assignment statement assigns a new value to a variable

Before Applesoft begins executing a program, it sets the values of all real and integer variables to 0 and the values of all string variables to the null string (that is, a string containing no characters). The program can then change the value of any variable at any time by executing an assignment statement.

The NEW, CLEAR, and RUN commands also reset all real and integer variables to 0 and all string variables to the null string.

An *assignment statement* consists of the optional keyword LET, followed by the name of the variable whose value is to be changed, an equal sign (=), and an expression denoting the new value to be assigned to that variable. The equal sign means "receives the value" or "is assigned the value"; it is often read simply as "gets" ("X gets Y plus 2"). The assignment statement means "evaluate the expression to the right of the equal sign and assign the resulting value to the variable named to the left of the equal sign." The variable will then continue to hold that value until it is changed by another assignment statement or is reset by a NEW, CLEAR, or RUN command. For example,

NEW **command:** see Section 1.2.1

CLEAR **command:** see Section 1.2.2

RUN **command:** see Section 1.2.4

Keyword LET is optional

```
LET Q = 27.4
```
—assign value 27.4 to real variable Q

```
LET D3 = J
```
—assign current value of real variable J to real variable D3; variable J unchanged

```
COUNT% = A + B
```
—assign current value of real variable A plus current value of real variable B to integer variable COUNT%; variables A and B unchanged

```
S9% = 35
```
—assign value 35 to integer variable S9%

```
NAME$ = "SAM"
```
—assign string value "SAM" to string variable NAME$

```
NAME$ = SAM$
```
—assign current value of string variable SAM$ to string variable NAME$; variable SAM$ unchanged

```
BOX(5) = 36
```
—assign value 36 to element 5 of real array BOX

```
J% = A%(N)
```
—assign current value of element N of integer array A% to integer variable J%; array A% and variable N unchanged

```
SHOP$(N) = "BAKERY"
```
—assign value "BAKERY" to element N of string array SHOP$

The keyword LET is optional in assignment statements. The statements

```
LET Q = 27.4
```

and

```
Q = 27.4
```

mean exactly the same thing.

2.3 Expressions

An *expression* is a formula describing a calculation for the computer to perform. It may involve any number of numeric variables and constants, together with *operators* specifying how the values of the variables and constants are to be combined. There are three kinds of operator that can be used in an Applesoft expression:

arithmetic operators: see Section 2.3.1

- *Arithmetic* operators combine two numeric values and produce a numeric result.

relational operators: see Section 2.3.2

- *Relational* operators compare two values and produce a logical (true-or-false) result.

logical operators: see Section 2.3.3

- *Logical* operators combine two logical values and produce a logical result.

Table 2-2 summarizes the various operators available in Applesoft.

2.3.1 Arithmetic Operators

Arithmetic operators combine numeric values to produce numeric results

Arithmetic operators combine two numeric values to produce a numeric result. There are five of them in Applesoft, corresponding to the

Table 2-2 Operators

Arithmetic Operators

+	addition
−	subtraction
*	multiplication
/	division
^	exponentiation

Relational Operators

=	equal to
<	less than
>	greater than
<=	less than or equal to
=<	
>=	greater than or equal to
=>	
<>	not equal to
><	

Logical Operators

AND	both true
OR	either or both true
NOT	is false

familiar operations of arithmetic: + (addition), − (subtraction), * (multiplication), / (division), and ^ (exponentiation). Here are some examples of their use:

```
3 + 4
```
—3 plus 4, yielding 7

```
+144
```
—plus 144 (a positive number)

```
X + Y
```
—the value of X plus the value of Y

```
23.7 - 11.4
```
—23.7 minus 11.4, yielding 12.3

```
50 - 75
```
—50 minus 75, yielding −25

```
-144
```
—minus 144 (a negative number)

```
SUM% - 2
```
—the value of SUM% minus 2

```
13 * 5
```
—13 times 5, yielding 65

```
6 * .25
```
—6 times .25, yielding 1.5

```
25 * QUARTERS%
```
—25 times the value of QUARTERS%

```
4.8 * COUNT(5)
```
—4.8 times the value of element 5 of array COUNT

```
18 / 6
```
—18 divided by 6, yielding 3

```
6 / 18
```
—6 divided by 18, yielding .333333333

```
DIST / TIME
```
—the value of DIST divided by the value of TIME

```
DOLLARS% / 100
```
—the value of DOLLARS% divided by 100

```
2 ^ 3
```
—2 to the 3rd power, yielding 8

```
3 ^ .5
```
—3 to the .5 power, yielding 1.73205081

```
X ^ J%
```
—the value of X raised to the power of the value of J%

Like most other computer languages, Applesoft uses an asterisk (*) instead of the letter X to represent multiplication.

What to Do with Fractions: Applesoft doesn't treat fractional numbers in the way that you are probably used to dealing with them. Most people would read the expression `3 3/4` as "three and three quarters." To Applesoft, however, the same expression would mean "thirty-three divided by four." (Applesoft ignores any spaces it finds in a number.)

It's easy to convert fractions to a form Applesoft will understand correctly. Just think of `3 3/4` as "three plus three divided by four". In other words, instead of typing

```
LET A = 3 3/4
```

type this instead:

```
LET A = 3 + 3/4
```

Applesoft will do the rest.

2.3.2 *Relational Operators*

Relational operators compare values and produce logical results

A *relational operator* tests for a relation between two values and produces a logical (true-or-false) result, depending on whether the particular relation does or doesn't hold between those two values. For example, the expression

```
A > B
```

means "the value of variable `A` is greater than that of variable `B`." If the current value of variable `A` is `5` and that of `B` is `3`, then the relation is true; if the value of `B` is `8`, the relation is false.

Relational operators are particularly useful in connection with the `IF...THEN` statement, discussed in Section 3.2.2.

`1` stands for *true*

`0` stands for *false*

The Truth about Applesoft: Applesoft actually uses numeric values to represent the logical values true and false: if the stated relation is true, the value of the relational expression is `1`; if the relation is false, the value of the expression is `0`. For example, if you type the statement

```
PRINT 6 > 12
```

in immediate execution, Applesoft will respond by displaying the number `0`, meaning "false"; if you type

```
PRINT 12 > 6
```

Applesoft will display the number `1`, meaning "true."

Applesoft has six relational operators (some of which can be written in more than one way): = (equal to), < (less than), > (greater than), <= or =< (less than or equal to), >= or => (greater than or equal to), and <> or >< (not equal to). Here are some examples of their use:

= means *equal to*

```
6 = 6
```
—6 equals 6, yielding 1 for true

```
6 = 12
```
—6 equals 12, yielding 0 for false

```
X = 2
```
—the value of X is equal to 2

```
NAME$ = "Ann"
```
—the value of NAME$ is equal to the string "Ann"

< means *less than*

```
6 < 6
```
—6 is less than 6, yielding 0 for false

```
6 < 12
```
—6 is less than 12, yielding 1 for true

```
PROBABILITY < .5
```
—the value of PROBABILITY is less than .5

> means *greater than*

```
6 > 6
```
—6 is greater than 6, yielding 0 for false

```
6 > 12
```
—6 is greater than 12, yielding 0 for false

```
AGE > 65
```
—the value of AGE is greater than 65

< = or =< means *less than or equal to*

```
6 <= 6
6 =< 6
```
—6 is less than or equal to 6, yielding 1 for true

```
6 <= 12
6 =< 12
```
—6 is less than or equal to 12, yielding 1 for true

```
A% <= 3 * B%
A% =< 3 * B%
```
—the value of A% is less than or equal to 3 times the value of B%

> = or = > means *greater than or equal to*

```
6 >= 6
6 => 6
```
—6 is greater than or equal to 6, yielding 1 for true

```
6 >= 12
6 => 12
```
—6 is greater than or equal to 12, yielding 0 for false

```
SALARY >= 20000
SALARY => 20000
```
—the value of SALARY is greater than or equal to 20000

< > or > < means *not equal to*

```
6 <> 6
6 >< 6
```
—6 is not equal to 6, yielding 0 for false

```
6 <> 12
6 >< 12
```
—6 is not equal to 12, yielding 1 for true

```
X <> Y
X >< Y
```
—the value of X is not equal to the value of Y

```
BANG$ >< "WHIMPER"
BANG$ <> "WHIMPER"
```
—the value of BANG$ is not equal to the string "WHIMPER"

2.3.3 *Logical Operators*

Logical operators combine logical values to produce logical results

A logical operator combines two logical (true-or-false) values and produces a logical result. There are three logical operators in Applesoft: AND, OR, and NOT. Here are some examples of their use:

AND yields true if *both* original expressions are true

```
6 <= 12 AND 6 >= 12
```
—6 is less than or equal to 12 and 6 is greater than or equal to 12; value is 0 for false

```
.25 <= R AND R < .75
```
—.25 is less than or equal to the value of R and the value of R is less than .75

OR yields true if *either or both* of the original expressions are true

```
6 <= 12 OR 6 >= 12
```
—6 is less than or equal to 12 or 6 is greater than or equal to 12; value is 1 for true

```
ANIMAL$ = "DOG" OR ANIMAL$ = "CAT"
```
—the value of ANIMAL$ is equal to the string "DOG" or the string "CAT"

NOT yields true if the original expression is *false*

```
NOT (6 <= 12)
```
—6 is not less than or equal to 12; value is 0 for false

```
NOT (YEAR% > 1950)
```
—the value of YEAR% is not greater than 1950

Notice that the OR operator doesn't correspond exactly to the way we often use the word "or" in everyday speech. When we say "A or B is true," we usually mean that exactly one of the two statements is true, but not both. The Applesoft OR operator produces a "true" value if either *or both* of the original expressions are true.

0 means *false*

Any nonzero value means *true*

More Truth about Applesoft: Applesoft's logical operators consider a numerical value of 0 to mean "false"; any numerical value other than 0 is taken to mean "true." The logical operators always yield a value of 1 for true or 0 for false.

Logical operators are particularly useful in connection with the IF...THEN statement, discussed in Section 3.2.2.

2.3.4 *Precedence of Operators*

Operators in Applesoft have an order of *precedence* that determines which operations are carried out first when they are combined in an expression. Table 2-3 lists the operators in descending order of precedence. Operators shown higher in the list are carried out before those lower down. Operators on the same line of the list have the same precedence, and are carried out from left to right within an expression.

Table 2-3 Precedence of Operators

Parentheses (innermost first)	( )
Signed arithmetic and logical NOT	+ − NOT
Exponentiation (powers of numbers)	^
Multiplication and division	* /
Addition and subtraction	+ −
Relational operators	= < > <= =< >= => <> ><
Logical AND	AND
Logical OR	OR

Notice in Table 2-3 that the operators + and − have higher precedence when used to represent the sign of a single number (as in +144 or −X) than when they stand for the addition or subtraction of two numbers.

To understand how Applesoft's precedence rules work, consider the expression

```
-2 * Z ^ 3 + Q / 5 - A * B
```

When Applesoft evaluates this expression, it begins by applying the first minus sign to the constant 2, obtaining a result of −2. Next it raises the value of Z to the 3rd power and multiplies the result by −2. Then it divides the value of Q by 5 and adds the result to that of the previous calculation. Finally, it multiplies the values of A and B and subtracts that result from the previous one.

For example, suppose the current values of the variables in the expression above are as follows: `Z = 2, Q = 10, A = 7, B = 4`. Then

```
- 2                              = -2
Z ^ 3          = 2 ^ 3          = 8
-2 * 8                          = -16
Q / 5          = 10 / 5         = 2
-16 + 2                         = -14
A * B          = 7 * 4          = 28
-14 - 28                        = -42
```

The value of the expression is `-42`.

Parentheses change order of evaluation

Parentheses can be used to change the normal order of precedence. For example:

```
-2 * Z ^ (3 + Q) / 5 - A * B
```
—value is `-3304.8`; `(3 + Q)` evaluated as a unit

```
-2 * Z ^ 3 + (Q / 5 - A) * B
```
—value is `-36`; `(Q / 5 - A)` evaluated as a unit

```
-2 * Z ^ 3 + Q / (5 - A * B)
```
—value is `-16.4347826`; `(Q / (5 - A * B))` evaluated as a unit

The original expression above is equivalent to the fully parenthesized expression

```
(((-2) / (Z ^ 3)) + (Q / 5)) - (A * B)
```

Helpful Hint: When you're unsure of the order of precedence, use parentheses to make sure the expression is evaluated in the order you intend.

Functions

2.4

function: a preprogrammed calculation that can be carried out on request

Functions are preprogrammed calculations that can be carried out on request. You can use them whenever you need to perform the same calculation repeatedly throughout a program. Whenever you *call* a function (request its execution), you must give it a particular value to operate on; this value is called the *argument* of the function.

Applesoft offers a variety of built-in functions, discussed in Section 2.4.1, for calculating common mathematical values such as logarithms, cosines, and square roots. Section 2.4.2 covers the built-in function RND, used for generating random numbers. In addition, you can define your own functions for the special needs of a particular program—see Section 2.4.3 for details.

2.4.1 Built-in Arithmetic Functions

Calling built-in functions

This section discusses the various built-in functions that Applesoft provides for calculating commonly used mathematical quantities. To call a built-in function, just type the name of the function followed by an expression in parentheses representing the argument value on which you want the function to operate. For example, suppose you need to calculate the square root of a number. Applesoft has a built-in function named SQR for this purpose; to find the square root of 3, write

```
SQR (3)
```

To find the square root of the value of variable X plus 2, write

```
SQR (X+2)
```

The ABS *Function*

ABS computes the absolute value

The built-in function ABS computes the absolute value of a number—that is, the positive numerical value of the number, without regard to its original sign. For example,

```
ABS (27)
```
—absolute value of 27; yields 27

```
ABS (-27)
```
—absolute value of −27; yields 27

```
ABS (36.8 - 23.3)
```
—absolute value of 36.8 minus 23.3; yields 13.5

```
ABS (23.3 - 36.8)
```
—absolute value of 23.3 minus 36.8; yields 13.5

```
ABS (C%(9))
```
—absolute value of element 9 of array C%

The SGN *Function*

SGN computes the sign of a number

The SGN function determines whether the value of its argument is positive, negative, or zero. It yields a result of 1 if the argument value is positive (greater than 0), −1 if the argument value is negative (less than 0), and 0 if the argument value is 0. For example,

```
SGN (27)              —sign of 27; yields 1 (positive)
SGN (-27)             —sign of -27; yields -1
                      (negative)
SGN (36.8 - 23.3)     —sign of 36.8 minus 23.3;
                      yields 1 (positive)
SGN (23.3 - 36.8)     —sign of 23.3 minus 36.8;
                      yields -1 (negative)
SGN (9 * 5 - 45)      —sign of 9 times 5 minus 45;
                      yields 0
SGN (SUM - 20)        —sign of SUM minus 20
```

The INT *Function*

INT computes the integer part of a number

integer: a whole number

INT yields the integer (whole-number) part of its argument value, with the fractional part (if any) discarded. Note that this function makes no attempt at rounding: that is, if the argument value is not an integer, INT yields the *next lowest integer,* not necessarily the nearest integer. For example,

```
INT (27)              —integer part of 27; yields 27
INT (36.8)            —integer part of 36.8; yields
                      36
INT (-7.9)            —integer part of -7.9; yields
                      -8
INT (-62.1)           —integer part of -62.1; yields
                      -63
INT (5 * PRICE)       —integer part of 5 times
                      PRICE
```

Rounding a Number: To round a numeric value to the *nearest* integer, first add .5 and then apply the INT function to the result. For example, to find the nearest integer to the current value of variable AGE, use the expression

```
INT (AGE + .5)
```

SQR computes the square root

The SQR *Function*

The SQR function computes the positive square root of its argument. For example,

```
SQR (169)          —square root of 169; yields 13
SQR (163.84)       —square root of 163.84;
                    yields 12.8
SQR (3)            —square root of 3; yields
                    1.73205081
SQR (X^2 + 9)      —square root of X squared plus
                    9
```

Square root of a negative number is an error

If you try to take the square root of a negative number, an ILLEGAL QUANTITY error will occur.

The SIN *Function*

SIN computes the sine

SIN computes the trigonometric sine of its argument. The argument must be expressed in radians. For example, assuming the value of the variable PI is 3.14159265,

```
SIN (PI / 3)       —sine of PI / 3 radians;
                    yields .866025403
SIN (1)            —sine of 1 radian; yields
                    .841470985
SIN (X^2 - Y^2)    —sine of X squared minus Y
                    squared
```

Arguments to trig functions must be in *radians*, not degrees

The argument you supply to the SIN function must be expressed in radians, not degrees. (There are 2π radians in a circle; one radian is equal to approximately 57.2957795 degrees.) For a formula you can use to convert from degrees to radians, see Section 2.4.3, "Defining Your Own Functions: The DEF FN Statement."

The COS *Function*

COS computes the cosine

COS computes the trigonometric cosine of its argument. The argument must be expressed in radians. For example, assuming the value of the variable PI is 3.14159265,

```
COS (PI / 3)       —cosine of PI / 3 radians;
                    yields .5
COS (1)            —cosine of 1 radian; yields
                    .540302306
```

```
COS (X^2 - Y^2)
```
—cosine of X squared minus Y squared

Arguments to trig functions must be in *radians,* not degrees

The argument you supply to the COS function must be expressed in radians, not degrees. (There are 2π radians in a circle; one radian is equal to approximately 57.2957795 degrees.) For a formula you can use to convert from degrees to radians, see Section 2.4.3, "Defining Your Own Functions: The DEF FN Statement."

The TAN *Function*

TAN **computes the tangent**

TAN computes the trigonometric tangent of its argument. The argument must be expressed in radians. For example, assuming the value of the variable PI is 3.14159265,

```
TAN (PI / 3)
```
—tangent of PI / 3 radians; yields 1.7320508

```
TAN (1)
```
—tangent of 1 radian; yields 1.55740772

```
TAN (X^2 - Y^2)
```
—tangent of X squared minus Y squared

Arguments to trig functions must be in *radians,* not degrees

The argument you supply to the TAN function must be expressed in radians, not degrees. (There are 2π radians in a circle; one radian is equal to approximately 57.2957795 degrees.) For a formula you can use to convert from degrees to radians, see Section 2.4.3, "Defining Your Own Functions: The DEF FN Statement."

The ATN *Function*

ATN **computes the arc tangent**

ATN computes the trigonometric arc tangent (inverse tangent) of its argument: that is, the angle whose tangent is equal to the given value. The result is expressed in radians. For example,

```
ATN (SQR(3))
```
—arc tangent of the square root of 3; yields 1.04719755 (= PI / 3) radians

```
ATN (1)
```
—arc tangent of 1; yields .785398163 radians

```
ATN (X^2 - Y^2)
```
—arc tangent of X squared minus Y squared

Result of ATN function is in *radians,* not degrees

The result produced by the ATN function is expressed in radians, not degrees. (There are 2π radians in a circle; one radian is equal to approximately 57.2957795 degrees.) For a formula you can use to convert from radians to degrees, see Section 2.4.3, "Defining Your Own Functions: The DEF FN Statement."

The EXP *Function*

EXP **computes the exponential**

The EXP function computes the mathematical exponential of its argument. The exponential is defined as the constant *e* raised to the power of the argument, where *e* = 2.718281828. For example,

```
EXP (3)              —e to the 3rd power; yields
                      20.0855369
EXP (LOG(10))        —e to the power of the natural
                      logarithm of 10; yields 10
EXP (A * T)          —e to the power A * T
```

EXP accurate to six places

Limited Accuracy: Although Applesoft will display the result of the EXP function to nine places, only the first six are actually reliable. For instance, in the first example above, the computed result of 20.0855369 should be interpreted simply as 20.0855.

The LOG *Function*

LOG **computes the natural logarithm**

LOG computes the natural logarithm of its argument (the logarithm to the base *e*, where *e* = 2.718281828.) For example,

```
LOG (10)             —natural logarithm of 10; yields
                      2.30258509
LOG (EXP(3))         —natural logarithm of e to the
                      3rd power; yields 3
LOG (SIN(THETA))     —natural logarithm of the sine of
                      THETA
```

Logarithm of a nonpositive number is an error

If you try to take the logarithm of a zero or negative number, an ILLEGAL QUANTITY error will occur.

2.4.2 *Generating Random Numbers: The* RND *Function*

RND **generates random numbers**

The built-in function RND produces random decimal numbers between 0 and 1. The behavior of this function depends on whether the argument you give it is positive, zero, or negative.

The simplest way to use RND is to give it a positive argument. RND will produce a different random number each time you call it with a

Positive argument produces a different random number each time

positive argument. The actual numeric value of the argument is ignored; only its sign is significant:

```
RND (1)          —yields .431448496
RND (1)          —yields .735966024
RND (99)         —yields .345445325
```

Zero argument repeats same result as previous call

If you give RND a zero argument, it will reproduce the same result as at the previous call:

```
RND (99)         —yields .270011996
RND (0)          —yields .270011996
RND (0)          —yields .270011996
RND (99)         —yields .139756248
RND (0)          —yields .139756248
```

Negative argument starts new, repeatable sequence

seed: the value used to begin a sequence of random numbers

Calling RND with a negative argument causes it to begin a new, repeatable sequence of random numbers. This is called *seeding* the random number generator; the particular negative value you use for the argument acts as a "seed" for the new sequence. Different seeds will produce different sequences, but each time you use the same seed you will get the same result. Subsequent calls to RND with positive arguments will then produce the same sequence of results:

```
RND (-1)         —yields 2.99196472E-08
RND (1)          —yields .738207502
RND (1)          —yields .272707136
RND (1)          —yields .299733446
RND (-5)         —yields 3.73720468E-08;
                    starts new sequence
RND (1)          —yields .407457285
RND (1)          —yields .463740324
RND (1)          —yields .387195686
RND (-1)         —yields 2.99196472E-08;
                    repeats same sequence as
                    before
RND (1)          —yields .738207502
RND (1)          —yields .272707136
RND (1)          —yields .299733446
```

Scientific Notation: The suffix E-08 in some of the random values listed above means "times 10 to the minus-8th power," and is an example of the *scientific notation* that Applesoft uses to display certain numbers. See Section I.2 for further details.

2.4.3 *Defining Your Own Functions: The* DEF FN *Statement*

```
DEF FN CUBE (X) = X * X * X
```

In addition to the built-in functions discussed in Sections 2.4.1 and 2.4.2, Applesoft gives you the ability to define your own functions for the special needs of a particular program. Defining your own functions can be a real time-saver: instead of writing out the same complex formula over and over again, you can simply define it once as a function, give it a name, and then refer to it by that name whenever you need it.

DEF FN **defines a new function**

To define a function of your own, use the DEF FN statement. This statement consists of the keywords DEF FN (for "define function") followed by the name of the function you're defining, the argument name enclosed in parentheses, an equal sign (=), and the formula defining the function. The examples below define functions to convert temperatures from Fahrenheit to Celsius and vice versa, and to convert angles from degrees to radians and vice versa, assuming that the value of the variable PI is 3.14159265:

argument: the value on which a function operates

```
10 DEF FN FTC (T) = (T - 32) * 5 / 9
                          —Fahrenheit to Celsius
20 DEF FN CTF (T) = T * (9 / 5) + 32
                          —Celsius to Fahrenheit
30 DEF FN DTR (A) = A * (PI / 180)
                          —degrees to radians
40 DEF FN RTD (A) = A * (180 / PI)
                          —radians to degrees
```

For example, the definition above for function FTC says "to convert from Fahrenheit to Celsius, take the value of the argument (T), subtract 32, multiply by 5, and divide by 9."

Formula limited to 239 characters

The formula defining a function must not exceed one program line (239 characters) in length.

The names you give to your functions must follow the same rules given in Section 2.1.1 for variable names: the name may be as long as you like (up to 239 characters), but Applesoft uses only the first two characters to distinguish one function from another. The argument variable in the function definition must be a real variable—integer and string variables (ending in % or $) are not allowed.

Argument must be a real variable

Don't begin function names with the same first two characters

Take care not to begin the names of different functions with the same two characters. Applesoft will consider the names CODFISH and COUNT, for example, to refer to the same function, since they both begin with the same two characters. If you try to define functions with these two names, the second definition will redefine the function, causing the first definition to be forgotten.

However, a program can have a function and an array beginning with the same two characters (or even having exactly the same name). This is because references to the function are written with the keyword FN (see below), but references to the array aren't. Thus Applesoft can tell that, for example,

```
FN COUNT (N)
```

is a call to the function named COUNT, whereas

```
COUNT (N)
```

is a reference to the array of the same name.

The DEF FN statement can be executed only from within a program; you can't use this statement in immediate execution.

Calling defined functions

To call a function that you've defined with DEF FN, type the keyword FN (for "function") followed by the name of the function and an expression in parentheses representing the argument value on which you want the function to operate. For example, using the functions defined above,

```
FN FTC (98.6)
```
—convert 98.6 degrees Fahrenheit to Celsius; yields 37

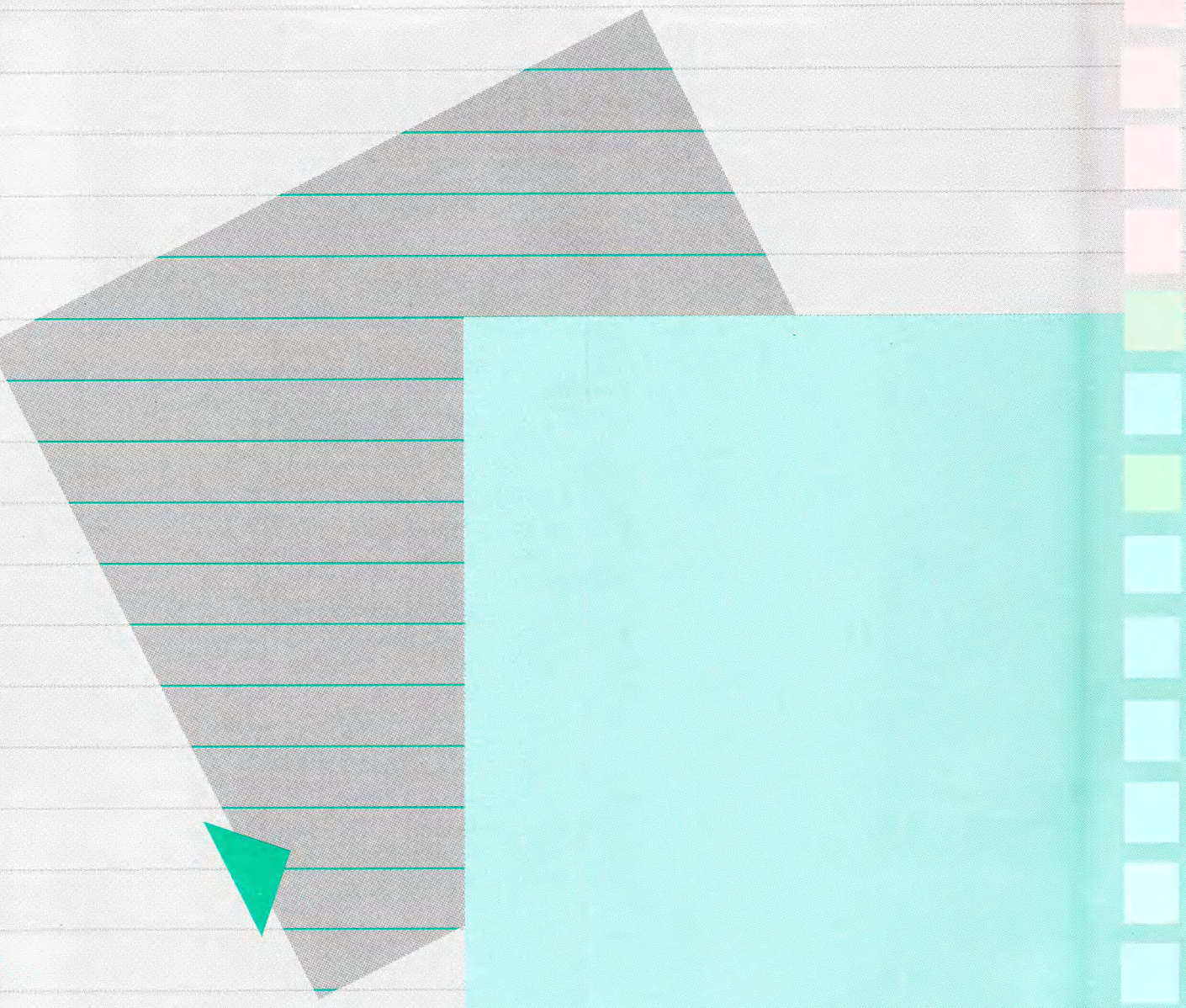
```
FN CTF (100)
```
—convert 100 degrees Celsius to Fahrenheit; yields 212

```
FN DTR (180)
```
—convert 180 degrees to radians; yields 3.14159265

```
FN RTD (PI / 2)
```
—convert PI / 2 radians to degrees; yields 90

Notice that the keyword FN must be used in calling your own defined functions, but not for built-in functions (see Section 2.4.1, "Built-in Arithmetic Functions").

Control Statements



Control Statements

Control statements determine the order of program execution

Ordinarily, Applesoft programs are executed sequentially, from the lowest-numbered line to the highest. *Control statements* allow you to *branch* to another part of the program: that is, to alter the normal order of execution and send control to a line of the program other than the next line in sequence. This ability to change the course of program flow is what gives computer programs their real power and flexibility.

unconditional branching: see Section 3.1

Section 3.1, "Unconditional Branching: The GOTO Statement," deals with the GOTO statement, which sends control unconditionally to a specified line of the program.

conditional branching: see Section 3.2

Section 3.2, "Conditional Branching," discusses conditional branching statements, which allow the program to decide what to do next by evaluating an expression or testing for a condition.

loops: see Section 3.3

Section 3.3, "Loops," covers statements that are used in loops (portions of a program that are executed many times repeatedly).

subroutines: see Section 3.4

Section 3.4, "Subroutines," deals with the very important subject of subroutines: sections of a program that can be executed on request from elsewhere in the program to perform some particular task.

error handling: see Section 3.5

Section 3.5, "Error Handling," describes Applesoft's facilities for detecting and dealing with error conditions that arise during the execution of a program.

program termination: see Section 3.6

Finally, Section 3.6, "Program Termination," covers the various ways of terminating (ending) the execution of a program.

3.1 Unconditional Branching: The GOTO Statement

```
GOTO 100
```

unconditional branch: a branch that does not depend on the truth of any condition

An *unconditional branch* sends control to a specified line of the program without reference to whether any particular condition holds. Applesoft has two statements that cause an unconditional branch: GOTO and GOSUB. The GOTO statement is described in this section; see Section 3.4.1 for a description of the GOSUB statement.

GOTO branches to a specified line number

The GOTO statement interrupts the normal sequential execution of program lines and forces execution to branch to (go to) a specified line number. The branch is *unconditional*: that is, it doesn't depend on the truth or falsity of any particular condition.

For example, consider the following program:

```
10 PRINT "HELLO"          —display the word HELLO
20 PRINT "THERE"          —display the word THERE
30 GOTO 10                —branch to line 10
40 PRINT "FRIEND"         —this line never executed
```

This program displays the word HELLO on the screen (line 10), displays the word THERE (line 20), and then (line 30) goes back to line 10 to repeat the process. The word FRIEND never gets displayed, because program execution never reaches line 40. Instead, the program simply repeats lines 10 to 30 indefinitely, displaying the words HELLO and THERE over and over again on the screen.

infinite loop: a section of a program that will repeat the same sequence of actions indefinitely

The program above contains an example of an infinite loop. To stop the program and regain control of the computer, press CONTROL-C.

If your program attempts to branch to a nonexistent line, or if a GOTO statement does not include a line number, an error message such as

```
?UNDEF'D STATEMENT ERROR IN 30
```

will appear, identifying the program line in which the error occurred. The program will stop and Applesoft will return to command level:

```
10 PRINT "HELLO"
20 PRINT "THERE"
30 GOTO 15                —branch to non-existent line
40 PRINT "FRIEND"
```

You Can't Branch to a Variable: If you attempt to use a variable instead of an actual line number to specify the line to which a branch should occur (as in `GOTO J`), Applesoft will always attempt to branch to line number 0, no matter what value the variable holds. If line 0 doesn't exist, an `UNDEF'D STATEMENT` error will occur:

```
 5 LET J = 10              —assign value to variable
10 PRINT "HELLO"
20 PRINT "THERE"
30 GOTO J                  —Applesoft will try to go to line
                            number 0
40 PRINT "FRIEND"
```

Conditional Branching

3.2

A *conditional branch* decides what action to take next, depending on the truth of a stated condition or on the value of an arithmetic expression. Applesoft has three statements that cause a conditional branch:

conditional branch: a branch that depends on the truth of a condition or the value of an expression

- `ON...GOTO` branches to one of a number of possible program lines, depending on the value of an arithmetic expression.
- `ON...GOSUB` branches to one of a number of possible subroutines, depending on the value of an arithmetic expression.
- `IF...THEN` either executes or skips one or more statements, depending on the truth of a stated condition.

`ON...GOTO` **statement:** see Section 3.2.1

`ON...GOSUB` **statement:** see Section 3.4.3

`IF...THEN` **statement:** see Section 3.2.2

The ON...GOTO Statement

3.2.1

```
ON X GOTO 150, 200, 310, 310, 150, 999
ON S% - 7 GOTO 300, 285, 900, 150
```

`ON...GOTO` **chooses where to branch depending on the value of an expression**

The `ON...GOTO` statement sends control to one of a list of line numbers, depending on the integer value of an arithmetic expression. The expression between the keywords `ON` and `GOTO` is evaluated; if the result is real it is truncated to an integer. If this value is between 1 and the number of line numbers in the list, program execution branches to the line number at the corresponding position in the list. (For example, if the integer value of the expression is 3, execution branches to the third line number in the list.) If the integer value of the expression is 0 or is greater than the length of the list, execution continues with the next statement following the `ON...GOTO`.

If **value out of range,** control proceeds sequentially

The following program illustrates the use of ON...GOTO:

```
 10 INPUT X                    —get number from keyboard
 20 ON X GOTO 150, 200, 310, 310, 150, 999
                               —decide where to go, depending on value of X
 30 PRINT "VALUE OUT OF RANGE, PLEASE
       RETYPE:"                —control comes here if X = 0 or X > 6
 40 GOTO 10                    —start again
150 PRINT "VALUE IS 1 OR 5"
                               —control comes here if X = 1 or X = 5
160 GOTO 10
200 PRINT "VALUE IS 2"
                               —control comes here if X = 2
210 GOTO 10
310 PRINT "VALUE IS 3 OR 4"
                               —control comes here if X = 3 or X = 4
320 GOTO 10
999 END                        —control comes here if X = 6
```

If the integer value of the expression between ON and GOTO is less than 0 or greater than 255, an ILLEGAL QUANTITY error will occur and program execution will halt.

3.2.2 *The* IF...THEN *Statement*

```
IF Z > 255 THEN END
IF H% - 23 < SM - TTL THEN K% =
   H% - 23 : H% = 0
IF X (I) = 12 THEN GOTO 325
IF A$ >< B$ THEN 300
IF (D > .05) AND NOT (E > .1) GOTO 2150
```

IF...THEN executes or skips, depending on the truth of a condition

The IF...THEN statement tests for the condition given between the keywords IF and THEN. If the condition is true, the statement or statements following THEN in the same program line are executed. If the condition is false, the remainder of the line following THEN is skipped and execution continues with the next program line in sequence.

When the statement following THEN is a GOTO, either (but not both) of the keywords THEN and GOTO may be omitted. The following statements are all equivalent:

```
IF X (I) = 12 THEN GOTO 325
IF X (I) = 12 THEN 325
IF X (I) = 12 GOTO 325
```

Notice that when the IF condition is false, program execution continues with the next *program line* (not the next statement) in sequence. No other statements in the IF line are carried out:

```
 10 LET J = 1 : K = 2
 20 LET A = 10          —A set to 10 here
 30 PRINT " J HOLDS "; J; " AND K
      HOLDS "; K
 40 IF A > 10 THEN J = 5: K = 10:
      GOTO 100          —A is not greater than 10; test
                          fails
 50 PRINT "THE VALUES OF J AND K ARE
      UNCHANGED."       —this message gets printed
 60 GOTO 999
100 PRINT "J NOW HOLDS "; J; " AND K
      HOLDS "; K        —this message not printed
999 END
```

When the program above is run, the IF test in line 40 will fail, the values of J and K will not be changed, and execution will continue with line 50. If line 20 were changed to

```
 20 LET A = 25
```

then the IF test in line 40 would succeed, the values of J and K would be changed, and control would branch to line 100.

0 means false

Any nonzero value means true

Using Numeric Values in IF...THEN: The IF...THEN statement considers a numeric value of 0 to mean "false"; any nonzero value is taken to mean "true." Thus you can write statements such as

```
IF J THEN GOSUB 400
```

which is equivalent to

```
IF J <> 0 THEN GOSUB 400
```

Recall also that Applesoft's relational and logical operators always yield a value of 1 for true and 0 for false. Thus you can combine numeric values with the logical operators: the statement

```
IF NOT J THEN GOTO 500
```

is equivalent to

```
IF J = 0 THEN GOTO 500
```

and

```
IF A AND B THEN END
```

is equivalent to

```
IF (A <> 0) AND (B <> 0) THEN END
```

Numeric values used in this way offer two advantages over the corresponding relational expressions:

- They take up less space in memory.
- They execute somewhat faster.

See Appendix G for further hints on making your programs more efficient.

Curious Parsing: Applesoft gets confused if the keyword THEN is immediately preceded by a variable name ending in the letter A. For example, the statement

```
IF J = BETA THEN 230
```

will be interpreted as

```
IF J = BET AT HEN230
```

causing a syntax error. This is because AT and THEN are both reserved words; in the example, the word AT is encountered first, so it is interpreted first. Such is life with Applesoft. You can get around the problem by using parentheses:

```
IF (J = BETA) THEN 230
```

3.3

Loops

loop: a sequence of statements executed repeatedly

pass: a single execution of a loop

A *loop* is a sequence of statements in a program that are executed repeatedly, often with the value of some variable being changed on each *pass* through the loop. Loops are fundamental to all computer programming: it's practically impossible to write any kind of useful or interesting program that doesn't include at least one loop.

index variable: a variable whose value changes on each pass through a loop; often called *control variable* or *loop variable*

The usual way of writing loops in Applesoft is with the FOR and NEXT statements. The FOR statement marks the beginning of the loop. It identifies the loop's *index variable* (the variable whose value changes on each pass) and gives the starting and ending values the index variable is to take on. Sometimes it also specifies the amount by which the value of the index variable is to change on each pass (see Section 3.3.1, "The FOR Statement," for details).

NEXT **statement:** see Section 3.3.2

The NEXT statement marks the end of the loop and causes the loop to be executed again for the next value of the index variable. When the loop has been executed once for each value of the index variable, as specified in the FOR statement, control "falls through" to the next statement following the NEXT statement. (That's right, "the next statement following the NEXT statement.")

Here's an example to show how loops work:

```
 5 PASS = 0                    —initialize pass count
10 FOR X = 3 TO 10             —execute loop once for each
                                  value of X from 3 to 10
20 PASS = PASS + 1             —count passes through the loop
30 PRINT "PASS #"; PASS
                               —display pass count
40 PRINT "INDEX = "; X
                               —display current value of index
                                  variable X
50 PRINT                       —display blank line (for
                                  neatness)
60 NEXT X                      —repeat loop for next value of X
70 PRINT "LOOP FINISHED"
                               —control comes here after last
                                  pass through loop
80 END
```

body: the statements in a loop between the FOR and NEXT statements

The loop begins with the FOR statement in line 10, which specifies that the loop is to be executed once for each value of index variable X from 3 to 10. Lines 20 to 50 form the *body* of the loop. The NEXT statement in line 60 marks the end of the loop and sends control back to line 20 for the next value of X. After the loop is executed for

the last time, with X set to the specified ending value of 10, X is increased to 11. Since this exceeds the ending value, control "falls through" the NEXT statement to line 70.

When the program above is executed, it will display the following results on the screen:

```
PASS #1
INDEX = 3
PASS #2
INDEX = 4
PASS #3
INDEX = 5
PASS #4
INDEX = 6
PASS #5
INDEX = 7
PASS #6
INDEX = 8
PASS #7
INDEX = 9
PASS #8
INDEX = 10
LOOP FINISHED
```

Be careful jumping out of loops

Loop Before You Leap: Exiting from the middle of a FOR/NEXT loop before the index variable reaches the ending value leaves Applesoft expecting a resolution that never comes. This is a dangerous practice that can cause unpredictable results in your program's execution. Don't write loops of the form

```
10 FOR INDEX = LOW TO HIGH
20 LET COUNT = COUNT + 1
30 IF COUNT = LIMIT THEN GOTO 100
                          —not recommended
40 NEXT INDEX
```

To be on the safe side, it's better to finish the loop this way:

```
30 IF COUNT = LIMIT THEN INDEX = HIGH:
     NEXT INDEX: GOTO 100
```

3.3.1 *The* FOR *Statement*

```
FOR Y = 1 TO 10
FOR MASS = 3.5 TO 7 STEP 1.5
FOR YEAR = 1980 TO 1960 STEP -4
FOR V = A + 2 TO 2*B - 3 STEP C / 2
```

FOR marks the start of a loop

The FOR statement marks the beginning of a loop, identifies the loop's index variable, and gives the starting and ending values of the index variable. It may also optionally specify the *step value*, the amount by which the value of the index variable is to change on each pass through the loop. If no step value is given, a value of 1 is understood.

index variable: a variable whose value changes on each pass through a loop; see Section 3.3, above

step value: the amount by which the index variable changes on each pass through a loop

In the example in Section 3.3 above, no step value was given, so the index variable X was incremented by 1 on each pass through the loop. If line 10 in the example were changed to

```
10 FOR X = 3 TO 10 STEP 2
```

—execute loop once for each value of X from 3 to 10 by 2

the program would produce the following output on the display screen:

```
PASS #1
INDEX = 3
PASS #2
INDEX = 5
PASS #3
INDEX = 7
PASS #4
INDEX = 9
LOOP FINISHED
```

The loop is executed four times, with the index variable taking on values of 3, 5, 7, and 9. At the end of the fourth pass, the index variable exceeds the specified ending value (9 plus 2 is 11, which is greater than the ending value of 10), so the loop ends and execution continues with the statement following the NEXT in line 60.

Body of loop always executed at least once

Applesoft Will Try Anything Once: Notice that the test to see whether the index variable exceeds the ending value is carried out at the *end* of the loop. This means the body of the loop will always be executed at least once. Even if the specified starting value is greater than the ending value, as in

```
10 FOR X = 10 TO 3        —starting value exceeds ending value
```

it won't be discovered until after the loop has been executed once (10 plus 1 is 11, which is greater than 3).

Step value may be negative

It's also possible to specify a negative step value:

```
10 FOR X = 10 TO 3 STEP -2
                          —negative step value
```

In this case the index variable will take values of 10, 8, 6, and 4. When the step value is negative, the loop ends when the index value becomes *less* than the ending value (4 plus −2 is 2, which is less than the ending value of 3). Notice that the starting and ending values have been reversed; if the statement read

```
10 FOR X = 3 TO 10 STEP -2
                          —starting value less than ending value
```

the loop would have been executed only once (3 plus −2 is 1, which is less than 10).

A step value of 0 will result in an infinite loop. To stop the program and regain control of the computer, press CONTROL-C.

Index variable must be a real variable

The index variable specified in a FOR statement must be a real variable. Attempting to use an integer variable, such as

```
10 FOR X% = 3 TO 10   —integer index variable
```

will cause a syntax error at run time. (However, the expressions for the starting, ending, and step values are unrestricted; any or all of these values may be specified by an integer variable).

3.3.2 *The* NEXT *Statement*

```
NEXT
NEXT INDEX
NEXT J,I
```

NEXT repeats execution of a loop

The NEXT statement marks the end of a loop and causes the loop to be repeated for the next value of the index variable, as specified in the corresponding FOR statement. When the value of the index variable becomes greater than the specified ending value (less than the ending value if the step value is negative), execution proceeds with the statement immediately following the NEXT statement.

Naming the index variable in a NEXT statement is optional; if you omit it, Applesoft will automatically repeat the most recently entered loop. If you're using nested loops, this means the innermost loop containing the NEXT statement will be repeated.

Helpful Hint: Leaving out the index variable in NEXT statements will make your programs run slightly faster:

```
10 FOR G = 1 TO 6
20 PRINT "WOW, MOM!"
30 NEXT                    —no index variable necessary
```

3.3.3 *Nesting of Loops*

nested loop: a loop contained within the body of another loop

FOR/NEXT loops can be *nested* one inside another to a maximum depth of ten levels. For example,

```
10 FOR A = 1 TO 3          —start of outer loop
20 FOR B = 1 TO 2          —start of inner loop
30 PRINT "A = "; A; ",B = "; B
                           —display values of index
                            variables
40 NEXT B                  —repeat inner loop
50 NEXT A                  —outer loop not repeated until
                            inner loop is finished
```

The inner loop (lines 20 to 40) is executed twice for each pass through the outer loop; the PRINT statement in line 30 is executed six times in all. This program will display the following on the screen:

```
A = 1, B = 1
A = 1, B = 2
A = 2, B = 1
A = 2, B = 2
A = 3, B = 1
A = 3, B = 2
```

No more than 10 levels of nesting

Although this example shows only two levels of nesting, Applesoft allows as many as ten levels (a loop inside a loop inside a loop . . . ten times). If you nest your loops to a depth greater than ten, your program will halt with the error message

```
OUT OF MEMORY
```

Don't cross loops

Nested loops must not cross each other—that is, each loop must be completely contained within the body of the next outer loop. Once a loop is started using a particular index variable, the corresponding NEXT must name the same index variable (if it names any at all). In the example above, if lines 40 and 50 were reversed

```
40 NEXT A          —attempt to repeat outer loop
50 NEXT B          —before inner loop is finished
```

the program would halt with an error because of the crossed loops.

Warning

Cross-looping is a second-degree misdemeanor punishable by five minutes in the penalty box and a NEXT WITHOUT FOR error. It will also melt your keyboard.

When two or more NEXT statements occur in a row, as in the example above, you can combine them into a single NEXT statement of the form

```
40 NEXT B,A        —repeat inner, then outer loop
```

Notice, however, that the index variables must be listed in the reverse order of their corresponding FOR statements, to avoid crossing loops. The statement

```
40 NEXT A,B        —whoops!
```

will reduce your keyboard to a puddle of plastic.

3.4

Subroutines

A subroutine is a section of a program that can be executed on request from another part of the program. Applesoft has four statements relating to subroutines:

GOSUB **statement:** see Section 3.4.1

- GOSUB directs control to a particular subroutine.

RETURN **statement:** see Section 3.4.2

- RETURN sends control back to the statement following the GOSUB that branched to the subroutine.

ON...GOSUB **statement:** see Section 3.4.3

- ON...GOSUB selects one of a number of possible subroutines, depending on the value of an arithmetic expression.

POP **statement:** see Section 3.4.4

- POP removes a return address from the top of the control stack (see the box below titled "How Subroutines Stack Up").

To *call* a subroutine (request its execution), branch to its first line with a GOSUB statement. GOSUB differs from an ordinary GOTO in that it "remembers" where in the program the subroutine was called from, so that control can return to that point when the subroutine is finished. The same subroutine can be called from many different places in the program; when the subroutine is finished, it sends control back to the statement following the proper point of call by executing a RETURN statement.

GOTO **statement:** see Section 3.1

point of call: the point in a program from which a subroutine is called

Here's an example to illustrate the idea:

```
 10 FOR Z = 1 TO 10 —execute loop 10 times
 20 LET X = INT (RND (1) * 100)
                    —generate a random integer be-
                      tween 0 and 99
 30 PRINT X " IS "; —display first part of message
 40 IF X < 50 THEN GOSUB 1000 : GOTO 60
                    —branch to subroutine at line
                      1000 if random number is
                      less than 50; on return, go to
                      line 60
 50 GOSUB 2000      —branch to subroutine at line
                      2000 if random number is
                      50 or greater
 60 PRINT "PASS #"; Z: PRINT
                    —count number of passes
                      through loop
 70 NEXT Z          —repeat loop
999 END             —end program
1000 PRINT "LESS THAN 50"
                    —print second part of message
                      for numbers less than 50
```

```
1010 RETURN                    —return to statement following
                                 point of call
2000 PRINT "MORE THAN 49"
                               —print second part of message
                                 for numbers greater than 49
2010 RETURN                    —return to statement following
                                 point of call
```

The loop in lines 10 to 70 generates a random integer between 0 and 99, then calls one of the two subroutines at lines 1000 and 2000, depending on the value of the random number. Each of the subroutines displays an appropriate message, then returns control to the statement following the point of call with a RETURN statement (lines 1010 and 2010). The program then displays a count of the number of passes through the loop and repeats the loop from the beginning. When the loop has been executed ten times, the program ends.

Control returns to *statement* (not line) following GOSUB

Notice that the RETURN statement returns control to the *statement* following the GOSUB statement, not just to the *line* following it. In line 40 of the example above, if the random number generated is less than 50, control is directed to the routine at line 1000. When execution returns from the subroutine, it will continue with the statement GOTO 60, branching around line 50.

Don't use the back door

Every subroutine should be regarded as a separate, indivisible unit of your program, which should be entered only with a GOSUB and exited only with a RETURN. Jumping into or out of the middle of a subroutine with an ordinary GOTO subverts Applesoft's orderly control stack mechanism (see "How Subroutines Stack Up," below) and causes the programmer to be in a state of sin. People who indulge in such odious practices should be ostracized from polite society.

nested subroutine call: a call to a subroutine from within another subroutine

Subroutine calls may be *nested:* that is, you can call one subroutine from inside another. Consider the following program:

```
  10 GOSUB 1000          —branch to first subroutine
  20 PRINT "BACK HOME AGAIN"
                         —this message displayed last
  30 END                 —prevent control from acci-
                           dently "falling into" a
                           subroutine
1000 PRINT "FIRST SUBROUTINE CALLED"
                         —this message displayed first
1010 GOSUB 2000          —branch to second subroutine
```

```
1020 PRINT "BACK AT FIRST SUBROUTINE"
                          —this message displayed third
1030 RETURN               —return to statement following
                           point of call (line 20)
2000 PRINT "SECOND SUBROUTINE CALLED"
                          —this message displayed
                           second
2010 RETURN               —return to statement following
                           point of call (line 1020)
```

Line 10 calls the first subroutine, at line 1000. This subroutine displays the first message on the screen, then (line 1010) calls the second subroutine at line 2000. The second subroutine displays its message, then returns control (line 2010) to the statement following the point of call in the first subroutine. The first subroutine then displays another message and returns control (line 1030) to the statement following its point of call. The final message is then displayed (line 20) and the program ends. The lines of the program are executed in the following order:

```
Line 10
Line 1000
Line 1010
Line 2000
Line 2010
Line 1020
Line 1030
Line 20
Line 30
```

The program produces the following output on the screen:

```
FIRST SUBROUTINE CALLED
SECOND SUBROUTINE CALLED
BACK AT FIRST SUBROUTINE
BACK HOME AGAIN
```

stack: a list in which entries are added or removed at one end only

return address: the point to which control returns on completion of a subroutine

push: to add an entry to the top of a stack

pop: to remove the top entry from a stack

How Subroutines Stack Up: Applesoft maintains a *control stack* to keep track of the *return addresses*—the points to which control is to return on completion—for all subroutines in progress. Each time a GOSUB is executed, the location of the statement following the GOSUB is *pushed* onto the top of the stack. When a RETURN statement is executed, the top entry is *popped* from the stack and control is directed to that point in the program. This arrangement ensures that control enters and leaves subroutines in LIFO (last-in-first-out) order.

Subroutine calls can be nested up to 25 levels deep: that is, you can GOSUB from a GOSUB from a GOSUB . . . 24 times. Attempting to go more than 25 levels deep will result in an OUT OF MEMORY error.

Actually, you're out of stack space, as opposed to program space. Since no rational BASIC program ever uses such complex nesting, this error usually means you've got a subroutine accidentally calling itself.

3.4.1 *The* GOSUB *Statement*

```
GOSUB 1000
```

GOSUB branches to a subroutine

The GOSUB (for "go to subroutine") statement is used to branch to a subroutine, saving a return address to which control can return when the subroutine is completed. The location of the statement immediately following GOSUB is pushed onto the control stack, and control is sent to the line number specified in the GOSUB statement. GOSUB differs from an ordinary GOTO in that it "remembers" where in the program the subroutine was called from, so that control can return to that point with a RETURN statement when the subroutine is finished.

control stack: see Section 3.4

GOTO statement: see Section 3.1

A GOSUB to a target line that doesn't exist will cause a message such as

```
?UNDEF'D STATEMENT ERROR IN 1350
```

to be displayed, identifying the line number in which the error occurred, and your program will come to an untimely halt.

3.4.2 *The* RETURN *Statement*

```
RETURN
```

RETURN returns control from a subroutine

The RETURN statement returns control from a subroutine to the statement following its point of call. The top entry is popped off the control stack and control is sent to that return address.

control stack: see Section 3.4

If the control stack is empty when RETURN is executed, your program will halt with the message

```
?RETURN WITHOUT GOSUB
```

3.4.3 *The* ON...GOSUB *Statement*

```
ON X GOSUB 150, 200, 310, 310, 150, 999
ON S% - 7 GOSUB 300, 285, 900, 150
```

ON...GOSUB chooses a subroutine depending on the value of an expression

The ON...GOSUB statement sends control to one of a list of subroutines, depending on the integer value of an arithmetic expression. The expression between the keywords ON and GOSUB is evaluated; if the result is real it is truncated to an integer. If this value is between 1 and the number of line numbers in the list, program execution branches to the subroutine at the corresponding position in the list. (For example, if the integer value of the expression is 3, execution branches to the subroutine beginning at the third line number in the list.) If the integer value of the expression is 0 or is greater than the length of the list, execution continues with the next statement following the ON...GOSUB.

If **value out of range,** control proceeds sequentially

The following program illustrates the use of ON...GOSUB:

```
 10 INPUT X                    —get number from keyboard
 20 ON X GOSUB 150, 200, 310, 310, 150,
      999                      —decide where to go, depending on value of X
 30 IF X = 0 OR X > 6 THEN PRINT "VALUE
      OUT OF RANGE, PLEASE RETYPE:"
                               —display message if X out of range
 40 GOTO 10                    —start again
150 PRINT "VALUE IS 1 OR 5"
                               —control comes here if X = 1 or X = 5
160 RETURN
200 PRINT "VALUE IS 2"
                               —control comes here if X = 2
210 RETURN
310 PRINT "VALUE IS 3 OR 4"
                               —control comes here if X = 3 or X = 4
320 RETURN
999 END                        —control comes here if X = 6
```

Compare the program above with the example given for the ON...GOTO statement in Section 3.2.1. The operation of ON...GOSUB is very similar to that of ON...GOTO, except that

ON...GOSUB "remembers" where in the program the subroutine was called from by pushing onto the control stack the location of the next statement following ON...GOSUB. Control can then return to that point with a RETURN statement when the subroutine is finished.

RETURN **statement:** see Section 3.4.2

If the integer value of the expression between ON and GOSUB is less than 0 or greater than 255, an ILLEGAL QUANTITY error will occur and program execution will halt.

3.4.4 *The* POP *Statement*

POP

POP **removes top entry from control stack**

control stack: see Section 3.4

The POP statement removes (*pops*) the top return address from the control stack without sending control to that point. This causes the next RETURN statement to send control back to the statement following the point of the second most recent subroutine call, instead of the most recent.

Here's an example illustrating the use of POP:

```
  10 GOSUB 1000          —branch to first subroutine
  20 PRINT "BACK HOME AGAIN"
                         —this message displayed last
  30 END                 —prevent control from acci-
                           dently "falling into" a
                           subroutine
1000 PRINT "FIRST SUBROUTINE CALLED"
                         —this message displayed first
1010 GOSUB 2000          —branch to second subroutine
1020 PRINT "BACK AT FIRST SUBROUTINE"
                         —this message never displayed
1030 RETURN              —this return never taken
2000 PRINT "SECOND SUBROUTINE CALLED"
                         —this message displayed
                           second
2005 POP                 —remove return address from
                           stack
2010 RETURN              —return to statement following
                           first subroutine's point of call
                           (line 20)
```

This program is identical to the one in Section 3.4 illustrating nested subroutine calls, except that a POP statement (line 2005) has been added to the second subroutine. The effect of the POP is to remove

the second subroutine's return address (line 1020) from the control stack, causing the RETURN in line 2010 to go back to the statement following the point of call of the *first* subroutine (line 20) instead. As a result, lines 1020 and 1030 are never executed, and the message BACK AT FIRST SUBROUTINE is never displayed. The lines of the program are executed in the following order:

```
Line  10
Line  1000
Line  1010
Line  2000
Line  2005
Line  2010
Line  20
Line  30
```

The program produces the following output on the screen:

```
FIRST SUBROUTINE CALLED
SECOND SUBROUTINE CALLED
BACK HOME AGAIN
```

If the control stack is empty when POP is executed, your program will halt with a RETURN WITHOUT GOSUB error.

Resist temptation

Programming Tip: Although it's sometimes tempting to try to get out of a tight programming situation by using POP, most good programmers avoid it, because it makes program flow really difficult to follow. If you find yourself becoming ensnared in convoluted code, 'tis a far better thing to redesign your program than to resort to the use of POP. See Chapter 8 for a tutorial on program planning.

3.5 *Error Handling*

Sometimes even the most carefully written program will come to an embarrassing halt at an inopportune moment because of an error. If you've never suffered an "error crash," you ain't a programmer. Applesoft's ONERR GOTO and RESUME statements provide a mechanism for detecting program errors as they occur and dealing with them from within your program. Using these statements, you can make your program display its own error messages or take any other action you consider appropriate, instead of coming to a sudden, screeching stop.

ONERR GOTO **statement:** see Section 3.5.1

RESUME **statement:** see Section 3.5.2

3.5.1 *The* ONERR GOTO *Statement*

```
ONERR GOTO 20000
```

ONERR GOTO allows program to handle errors

The ONERR GOTO statement turns off Applesoft's normal error handling and replaces it with an error-handling subroutine in your program. After this statement is executed, program errors will no longer stop the program, but will instead transfer control to the error routine beginning at the specified line number.

error code: a number representing a type of error

Error code stored at **location 222**

Before sending control to the error routine, Applesoft stores an error code identifying the type of error at a special location in the comptuter's memory, location 222. The error routine can then look at the contents of this location with the PEEK function and decide what action to take, depending on the error. Table 3-1 lists the possible error codes and their meanings. See Appendix E, "Error Messages," for further information on the conditions that cause each type of error.

meanings of errors: see Appendix E

Table 3-1 Error Codes

Code	Meaning	Code	Meaning
0	NEXT without FOR	120	Redimensioned array
16	Syntax	133	Division by zero
22	RETURN without GOSUB	163	Type mismatch
42	Out of data	176	String too long
53	Illegal quantity	191	Formula too complex
69	Overflow	224	Undefined function
77	Out of memory	254	Bad response to INPUT statement
90	Undefined statement	255	CONTROL-C interrupt attempted
107	Bad subscript		

To prevent an error from interrupting the program, the ONERR GOTO statement must be executed before the error occurs. If you're using ONERR GOTO, it's a good idea to make it one of the first lines in your program, as in the following example:

```
   10 ONERR GOTO 21500
                              —establish error routine at line
                                21500
                .
                .
                .
21500 LET EC=PEEK (222)
                              —get error code
21510 IF EC <> 255 THEN 21550
                              —branch if not CONTROL-C
21520 PRINT "SORRY--PROGRAM CAN'T BE
        STOPPED WITH CONTROL-C"
                              —if user pressed CONTROL-C,
                                display special message
21530 RESUME                  —and resume program
21550 PRINT "UNANTICIPATED ERROR,
        CODE "; EC            —on any other error, display
                                general message
21560 STOP                    —and halt
```

CONTROL-C: see Section 1.3.2

The program above uses its own error-handling routine to prevent the user from interrupting execution by pressing CONTROL-C. Line 10 turns off Applesoft's normal error handling and substitutes instead the program's own error routine, beginning at line 21500. If an error later occurs, the first thing the error routine does (line 21500) is get the error code from memory location 222 to find out what type of error occurred. The error code is assigned to variable EC to make it easier to handle. Line 21510 tests for an error code of 255, meaning "CONTROL-C interrupt attempted" (see Table 3-1). If the error is a CONTROL-C, the message

```
SORRY--PROGRAM CAN'T BE STOPPED WITH
  CONTROL-C
```

RESUME **statement:** see Section 3.5.2

is displayed on the screen (line 21520) and control is sent back to the point of the error with the RESUME statement in line 21530.

If the error isn't a CONTROL-C, the IF...THEN test in line 21510 sends control to line 21550. Since the error routine has no special action to take for any of these other errors, and since Applesoft's nor-

mal error messages are not being displayed, the error routine just displays a general error message such as

```
UNANTICIPATED ERROR, CODE 16
```

(for a syntax error) and stops the program.

Warning

Once an ONERR GOTO statement has been executed, ordinary error messages will not be displayed and the program will not stop if an error is detected. If your program's own error routine doesn't take some appropriate action (such as stop) for every possible error code, the program may hang indefinitely or exhibit other forms of deviant behavior. Make sure your error routine tells the computer what to do in all possible cases of error; see the following box for suggestions.

Cover all the bases

More Peeking: In the program above, the general error message displayed in line 21550 would be more useful if it included the line number where the error occurred as well as the error code itself. Through the magic of the PEEK function, the following two lines (replacing line 21550 of the original example) will do the trick:

PEEK **function:** see Section 7.1.1

```
21550 EL = PEEK (219) * 256 + PEEK (218)
                                  —get error line
21555 PRINT "UNANTICIPATED ERROR, CODE ";
        EC; ", IN LINE "; EL
                                  —display general error message
```

For more information...

For an even nicer way of handling unanticipated errors, see Section 3.5.3, "Restoring Normal Error Handling." See Appendix F, "Peeks, Pokes, and Calls," for more astounding feats of sorcery and witchcraft you can perform at home.

No ONERR GOTO in immediate execution

The ONERR GOTO statement can be executed only from within a program; you can't use this statement in immediate execution.

3.5.2 *The* RESUME *Statement*

```
RESUME
```

RESUME **returns control from an error routine**

The RESUME statement returns control from an error-handling routine to the statement in which the error occurred. It should be used only in error routines, and should never be encountered in the normal flow of control.

Warning

If Applesoft encounters a RESUME statement without an error having occurred, the program may stop or hang indefinitely, or other unpredictable but probably unpleasant events may transpire.

Warning

Notice that RESUME sends control back to the same statement that caused the error in the first place. If the same error occurs again, the program may hang in an infinite loop. Similarly, if an error occurs within the error-handling routine itself, RESUME will cause the program to hang.

Don't leave a mess!

Cleaning the Stack: When an error occurs while an ONERR GOTO statement is in effect, Applesoft pushes certain information onto its internal control stack before transferring control to the error routine. When you leave the error routine with a RESUME statement, these control codes are automatically popped off the stack. But if the error routine ends with a GOTO instead of a RESUME, the control codes will remain behind on the stack, probably causing the world to end with a whimper later on. To avert a global catastrophe, always "clean up" the stack by uttering the magical incantation

```
CALL -3288
```

before leaving an error routine with a GOTO statement.

control stack: see Section 3.4

GOTO **statement:** see Section 3.1

CALL **statement:** see Section 7.1.3

Warning

The RESUME statement should be executed only from within a program. Attempting to use this statement in immediate execution may cause a syntax error, cause the system to hang, or begin executing an existing or even a deleted program.

Don't use RESUME in immediate execution!

3.5.3 *Restoring Normal Error Handling*

POKE 216,0 **restores normal error handling**

You can restore Applesoft's normal error-handling mechanism by using the POKE statement:

```
POKE 216,0
```

POKE **statement:** see Section 7.1.2

After executing this statement, Applesoft will go back to stopping the program when an error occurs and displaying its usual error messages.

Apple IIe Monitor program: see *Apple IIe Reference Manual*

One use of this technique is to prevent your program from hanging or falling into the Monitor in case an error occurs in the error-handling routine itself. You can do this by restoring normal error handling with `POKE 216,0` at the beginning of your error routine, then reactivating the error routine with `ONERR GOTO` before returning to the main program. Here's another version of the example program of Section 3.5.1 that illustrates this technique:

```
10 ONERR GOTO 21500
                              —establish error routine at line
                                21500
                .
                .
                .
                .
                .
                .
21500 POKE 216,0              —restore normal error handling
21505 LET EC=PEEK (222)
                              —get error code
21510 IF EC <> 255 THEN 21540
                              —if not CONTROL-C, resume
                                program under normal error
                                handling
21520 PRINT "SORRY--PROGRAM CAN'T BE
        STOPPED WITH CONTROL-C"
                              —if user pressed CONTROL-C,
                                display special message,
21530 ONERR GOTO 21500
                              —reactivate this error routine,
21540 RESUME                  —and resume program
```

This program also illustrates another application of `POKE 216,0`. Notice that if the error is anything other than a CONTROL-C interrupt (code `255`), the `IF...THEN` test in line `21510` sends control directly to the `RESUME` statement in line `21540`, without executing the `ONERR GOTO` in line `21530`. The effect of this is to re-execute the statement containing the original error, but with Applesoft's normal error handling still in effect. This will cause the same error to occur again, but this time Applesoft will display its normal error message and halt the program. Thus CONTROL-C is the only error that gets special handling; all other errors produce the same results as if there were no special error routine.

3.6 Program Termination

The STOP and END statements are used to halt the execution of a program. The only difference between them is that STOP displays a message giving the number of the line at which execution was halted; this information is useful primarily for debugging purposes. END simply stops the program without any message, and is usually used at a program's natural finishing point.

debugging: finding and correcting errors in a program

3.6.1 The STOP Statement

```
STOP
```

STOP halts the program and displays a message

The STOP statement halts execution of the program and displays a message giving the number of the program line in which the STOP occurs. For example, the line

```
115 STOP
```

displays the message

```
BREAK IN 115
```

Applesoft returns to its command level, allowing you to enter new lines, examine or change the values of variables, and so on. You can then resume the execution of the program using the CONT command.

CONT command: see Section 1.3.3

3.6.2 The END Statement

```
END
```

END halts execution quietly

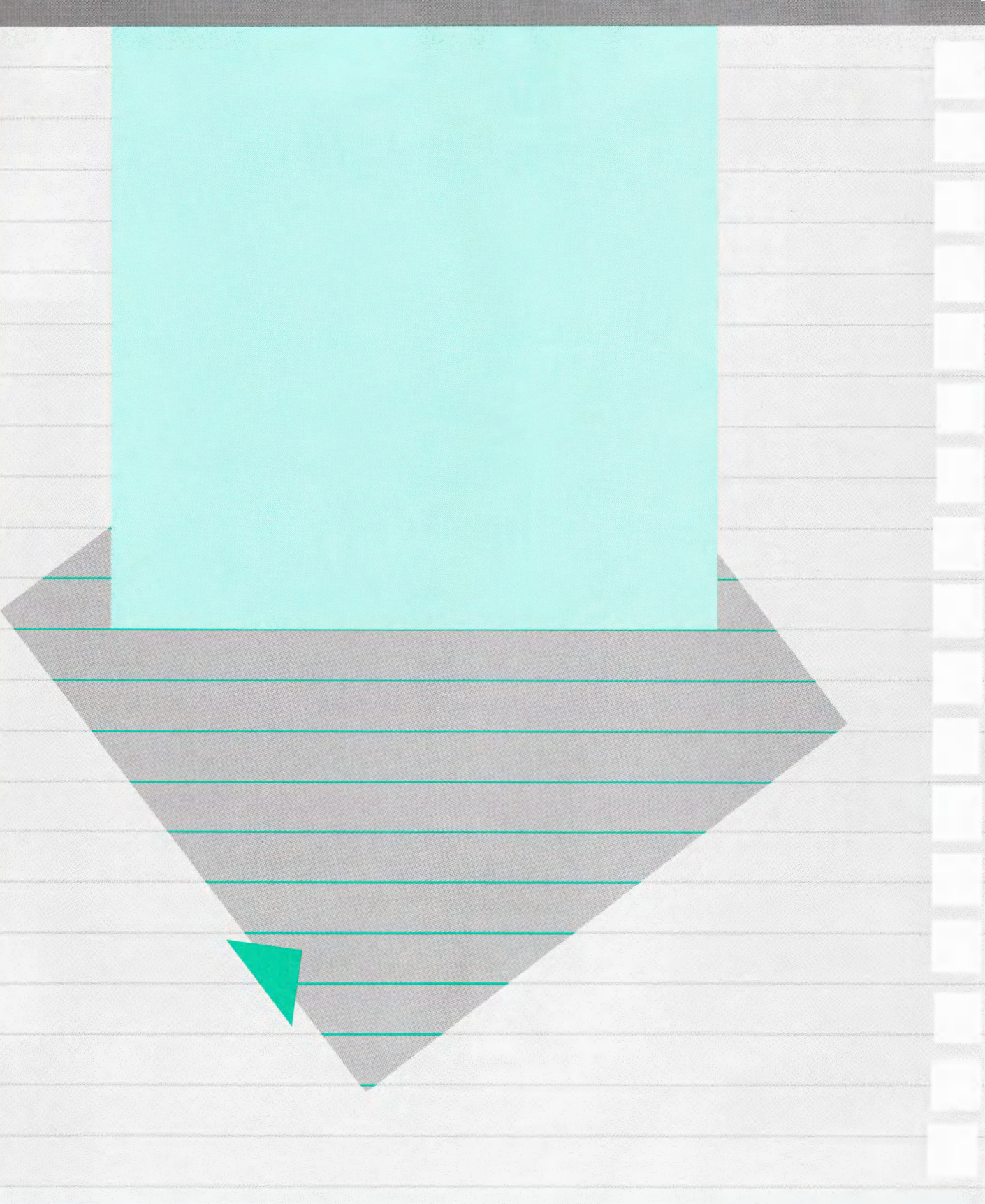
The END statement halts execution of the program and returns control to Applesoft's command level. No message is displayed on the screen; program execution just stops quietly.

```
999 END
```

END optional at end of program

An END statement is purely optional at the end of a program. The program will end by itself, even without an END statement, when it runs out of statements to execute.

Arrays and Strings



Arrays and Strings

This chapter discusses two important forms of data that Applesoft programs can operate on: arrays and strings. Both topics were treated briefly in Chapter 2, "Variables and Arithmetic," but are covered in more detail here.

arrays: see Section 4.1

Section 4.1, "Arrays," deals with collections of related information of any type (real, integer, or string), referred to by the same name and distinguished by means of numerical subscripts.

strings: see Section 4.2

Section 4.2, "Strings," describes Applesoft's facilities for manipulating strings of characters such as words or names: comparing them, concatenating (chaining) them together, taking them apart, and converting them to and from numeric values.

4.1 Arrays

array: a collection of variables referred to by the same name

An *array* is a collection of variables referred to by the same name, usually holding a collection of data items that are related to each other in some logical or systematic way. The individual variables in the array are called its *elements,* and are distinguished from one another by means of identifying index numbers called *subscripts.*

element: one of the individual variables in an array

An array can be of any type: integer, real, or string. Array names follow the same rules as simple variable names of the same type. To refer to a particular element of an array, write the array name followed by one or more subscripts, separated by commas and enclosed in parentheses. The subscripts refer to the position of the desired element within the array:

simple variable: a variable that is not an element of an array

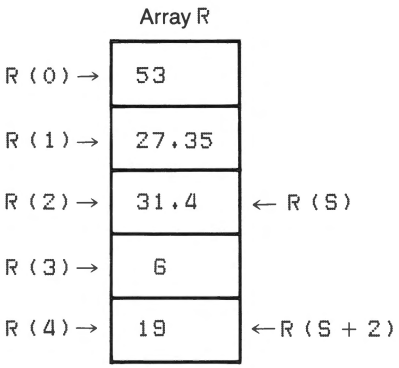
```
Q (6)              —element 6 of real array Q
FIGURE% (N)        —element N of integer array
                     FIGURE%
```

```
NAME$ (J - 3)          —element J - 3 of string
                         array

COUNT (SUM%, 2)        —element (SUM%, 2) of real
                         array COUNT
```

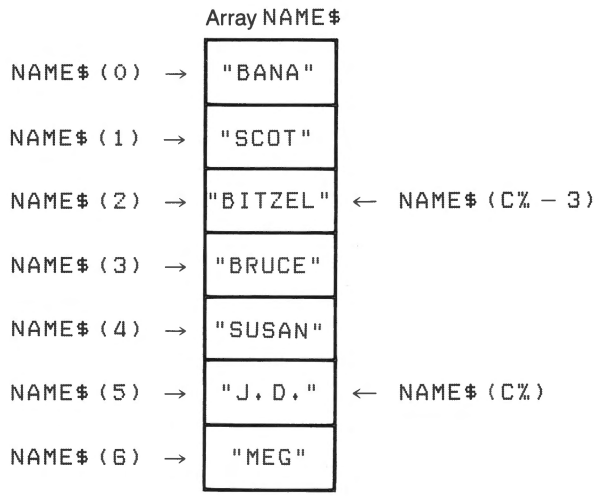
Figure 4-1 shows a real array named R with five elements, numbered 0 to 4. Element R(0) (pronounced "R-sub-zero") holds the value 53, R(1) holds 27.35, and so on. If the value of variable S is 2, then the expression R(S) refers to element R(2), whose value is 31.4, and the expression R(S + 2) refers to element R(4), which holds the value 19.

Figure 4-1 A Real Array



Another example is shown in Figure 4-2, this time a string array named NAME$ with seven elements, numbered 0 to 6. Element NAME$(1) holds the string value "SCOT", NAME$(3) holds the value "BRUCE", NAME$(6) holds "MEG", and so on. If the value of variable C% is 5, then the expression NAME$(C%) refers to element NAME$(5), whose value is "J.D.", and the expression NAME$(C% - 3) refers to element NAME$(2), which holds the value "BITZEL".

Figure 4-2 A String Array



4.1.1 *The* DIM *Statement*

```
DIM R (4)
DIM TITLE$ (100)
DIM H5 (J%)
DIM MARK% (3, C / 5, P + 2)
```

DIM defines the size of an array

dimension: the maximum size of one of the subscripts of an array

The DIM (for "dimension") statement defines the size of an array and allocates memory space for its elements. The expressions in parentheses following the array name give the *dimensions* of the array. There may be from one to 88 dimensions (see Section 4.1.2, "Multi-dimensional Arrays").

Once an array has been defined in a DIM statement, any reference to that array with a different number of subscripts, or with a subscript that exceeds the maximum specified for that dimension in the DIM statement, will cause the program to halt with the message

```
?BAD SUBSCRIPT ERROR
```

Available memory limits size of arrays

Arrays are limited in size by the amount of available memory. See Section H.2, "Applesoft Memory Allocation," for detailed information on the amount of space required by each type of array.

Subscripts start from 0, not 1

Since array subscripts in Applesoft begin with 0 (not 1), there is actually one more than the specified number of subscripts in each dimension. For instance, the array TITLE$ defined in the second example above has 101 (not 100) elements. In the definition

```
DIM TEST (12, 3, 5)      —array TEST has 13*4*6 =
                            312 elements
```

array TEST has 312 elements (13 times 4 times 6), not 180 (12 times 3 times 5) as you might expect.

When Applesoft encounters a reference to an array that has not yet been defined in a DIM statement, it automatically allocates space for 11 subscripts (0 to 10) in each dimension of the array. Later attempting to redefine the same array with a DIM statement will cause an error stop with the message

```
?REDIM'D ARRAY ERROR
```

Defining the same array in more than one DIM statement, or executing the same DIM statement twice, will produce the same message.

4.1.2 *Multidimensional Arrays*

The examples shown in Figures 4-1 and 4-2 are both one-dimensional arrays. Actually, arrays in Applesoft may have as many as 88 dimensions, subject to the amount of memory available. Arrays of 88 dimensions aren't terribly useful, but those of two and three dimensions often are.

Figure 4-3a shows an example of a two-dimensional array named EGGS, which has been defined by the DIM statement

```
DIM EGGS (1, 5)
```

Figure 4-3a A two-dimensional array

Array EGGS

	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5
Row 0 →	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
Row 1 →	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)

For the newly perplexed, a metaphor may be helpful. Think of the array as an empty egg carton. On the outside is written the word EGGS. When you open the egg carton, there are a dozen cup-like indentations where the eggs go—two rows of six cups each—corresponding to the elements of the array. Each of the cups is identified by a row number, 0 or 1, and a column number from 0 to 5 (we're dealing with strange chickens here).

Don't forget subscript 0!

Now suppose you place three eggs in the egg carton, in elements (0,2), (0,5), and (1,3):

```
LET EGGS (0, 2) = EGG
LET EGGS (0, 5) = EGG
LET EGGS (1, 3) = EGG
```

Figure 4-3b

Array EGGS

	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5
Row 0 →			EGG			EGG
Row 1 →				EGG		

Figure 4-3b shows the result. You might also elect to use your egg

carton to hold small change. If you put a nickel in position (0,1), a dime in position (1,1), and a quarter in position (1,4),

```
LET EGGS (0, 1) = 5
LET EGGS (1, 1) = 10
LET EGGS (1, 4) = 25
```

Figure 4-3c

Array EGGS

	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5
Row 0 →		5	EGG			EGG
Row 1 →		10		EGG	25	

your carton would look like Figure 4-3c.

Scrambled metaphor

Actually, of course, you can't store eggs in your Applesoft arrays, only numbers and strings—but after all, metaphors aren't always eggsact.

4.2 Strings

string: a sequence of text characters

A *string* is a sequence of text characters (letters, digits, and punctuation marks). Just as you can write numeric constants such as 27 and 2,236 in your Applesoft programs, you can write *string constants* by enclosing the characters in the desired string between double quotation marks:

String constants enclosed in double quotation marks

```
"ON SALE FOR $49.95"
"Truth is impervious to hissing"
"H2SO4"
"???"
```

Lowercase OK in string constants

Even though Applesoft doesn't understand lowercase letters when you use them in keywords, it will allow you to use them in a string constant, as the second example above shows.

null string: a string containing no characters

A string can contain from 0 to 255 characters; when it contains no characters at all, it is called a *null string.* Two quotation marks with nothing between them denote the null string:

```
""
```

—a string with no characters

String variable names end with $

A string variable can hold any string as its value. Its name must end with a dollar sign ($). Some legal string variable names are

```
TITLE$
G2$
D$
```

String variables preset to null string

Until they are given some other value with an assignment statement, all string variables are preset to the null string.

4.2.1 *Comparison of Strings: The ASCII Code*

character code: a number used inside the computer to represent a text character

ASCII: American Standard Code for Information Interchange; see Appendix C

The characters in a string are represented inside the computer in the form of numbers from 0 to 127. The correspondence between these internal *character codes* and the characters they represent is based on a nationwide computer-industry standard called the American Standard Code for Information Interchange, or ASCII (pronounced "asky"). For instance, ASCII code 65 represents the uppercase letter A, 112 represents a lowercase p, 52 represents the digit 4, 43 represents a plus sign (+), and so on. For a complete table of ASCII character codes and the characters they represent, see Appendix C, "ASCII Character Codes."

relational operators: see Section 2.3.2

Like numbers, strings can be compared with each other using the relational operators. The result of the comparison is based on the II codes of the characters in the strings. Applesoft looks for the first non-identical characters in the two strings and compares them by ASCII code to decide which is greater. For example, the character F (ASCII 70) is considered greater than the character D (ASCII 68) but less than the character H (ASCII 72). If one string is longer but begins with all the same characters as the other string, the longer string is considered greater. For example,

```
"A"     is less than    "B"
"AA"    is less than    "AB"
"AB"    is less than    "BA"
"AB"    is less than    "ABC"
```

String comparisons can be used for conventional **alphabetical order . . .**

Since letters of the alphabet are represented by consecutive codes in the ASCII table, comparisons between strings of alphabetic letters can be used to place the strings in conventional alphabetical order. For example,

```
"E"          is less than    "F"
"ED"         is less than    "EDGAR"
"EDGAR"      is less than    "EDWARD"
"EDWARD"     is less than    "EDWARDS"
"EDWARD"     is less than    "FRANK"
```

. . . but watch out!

There are a few surprises, however: since uppercase letters precede lowercase letters in the ASCII chart,

```
"Zebra"      is less than    "aardvark"
```

And since strings are compared strictly character by character,

```
"48"         is less than    "5"
```

VAL **function:** see Section 4.2.5

If you want to compare two strings consisting of digits according to the numbers they represent, use the VAL function.

4.2.2 *The* LEN *Function*

LEN **gives length of a string**

concatenation: see Section 4.2.3

The LEN (for "length") function counts the number of characters in a string. The argument may be a string constant, a string variable, or a concatenation of two or more strings. For example,

```
LEN ("APPLE")          —length of the string
                         "APPLE"; yields 5

LEN (SAMPLE$)          —length of the string contained
                         in variable SAMPLE$

LEN (A$ + "***" + B$)
                       —length of the concatenation of
                         variable A$, string "***",
                         and variable B$
```

Using LEN, you can assign the length of a string to a numeric variable and then use it in further operations:

```
10 LET N% = LEN ("MY HEART SOARS LIKE A
     HAWK.")
20 PRINT "THERE ARE "; N%; " CHARACTERS
     IN THE STRING."
```

When executed, this program will display the following output on the screen:

```
THERE ARE 27 CHARACTERS IN THE STRING.
```

If you concatenate two or more strings with a combined length of more than 255 (the maximum allowable string length), your program will halt with the message

```
?STRING TOO LONG ERROR
```

Instead of writing

```
LEN (A$ + B$ + C$)
```

it's safer to use

```
LEN (A$) + LEN (B$) + LEN (C$)
```

4.2.3 *Concatenation of Strings*

concatenate: to combine two or more strings into a single, longer string

Concatenation means "chaining together." To concatenate two or more strings is to join them together into a new string containing all the characters of the original strings combined. This operation is represented in Applesoft by a plus sign (+):

```
"BORIS" + " AND " + "NATASHA"
```
—concatenation of the strings "BORIS", " AND ", and "NATASHA"; yields the string "BORIS AND NATASHA"

```
F$ + C$
```
—concatenation of the contents of string variables F$ and C$

```
H$ + "RATS!"
```
—concatenation of the contents of string variable H$ with the string constant "RATS!"

LEFT$ **function:** see Section 4.2.4

```
H$ + LEFT$(C$, 4)
```
—concatenation of the contents of string variable H$ with the leftmost four digits of the contents of string variable C$

The program

```
10 LET NAME$ = "CHARLIE"
                          —set victim's name
20 LET TITLE$ = "DEAR " + NAME$ + ","
                          —form salutation
30 PRINT TITLE$           —print salutation
40 PRINT "HAVE WE GOT A SALE!"
                          —print rest of message
```

will display the output

```
DEAR CHARLIE,
HAVE WE GOT A SALE!
```

on the screen. The program

```
10 LET A$ = "GOOD "   —assign value to string variable
20 LET A$ = A$ + "GRIEF!"
                      —extend string with
                        concatenation
30 PRINT A$           —display result
```

will display

```
GOOD GRIEF!
```

Result must not exceed 255 characters

If the result of a concatenation operation is a string more than 255 characters in length, the program will halt with the error message

```
?STRING TOO LONG ERROR
```

LEN **function:** see Section 4.2.2

You can test how long the result of a concatenation will be beforehand by using the LEN function. For example:

```
10 LET A$ = "HAPPY DAYS "
20 LET L1 = LEN (A$) —how many characters in A$?
30 LET B$ = "ARE HERE AGAIN"
40 LET L2 = LEN (B$) —how many characters in B$?
50 IF (L1 + L2) < 256 THEN LET A$ =
     A$ + B$          —if the combined lengths of A$
                        and B$ are less than 256,
                        combine the two strings into
                        A$
```

+ on strings doesn't mean addition!

Don't confuse the concatenation of strings with the addition of numbers, even though both are represented in Applesoft by the same symbol (+). The value of the expression

```
12 + 34
```

is the number 46; the value of the expression

```
"12" + "34"
```

is the string "1234". If you want to add two strings consisting of digits according to the numbers they represent, use the VAL function.

VAL function: see Section 4.2.5

4.2.4 Substring Functions

substring: a string that is part of another string

Applesoft has three built-in functions for extracting *substrings* from a string:

- LEFT$ extracts a substring from the beginning of a string.
- MID$ extracts a substring from anywhere in a string.
- RIGHT$ extracts a substring from the end of a string.

The LEFT$ Function

LEFT$ extracts a substring from the beginning of a string

LEFT$ extracts a specified number of characters from the beginning (left end) of a string. The LEFT$ function takes two arguments, separated by a comma: the string from which the characters are to be taken and the number of characters desired. For example,

```
LEFT$ ("THIS IS IT!", 4)
```
—first 4 characters of the string "THIS IS IT!"; yields "THIS"

```
LEFT$ (NAME$, C + 2)
```
—first C + 2 characters of the contents of string variable NAME$

Real arguments converted to integers

If the value you give for the number of characters in the substring is a real number, LEFT$ truncates it to the next lowest integer. If the value specified is greater than the length of the string, Applesoft returns the entire string; no extra characters are added.

The number of characters requested must be between 1 and 255 or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

If you omit the dollar sign ($) from the function name LEFT$, Applesoft will treat LEFT as an arithmetic variable name, causing an error stop with the message

```
?TYPE MISMATCH ERROR
```

The MID$ *Function*

MID$ extracts a substring from anywhere in a string

MID$ (for "middle") extracts a specified number of characters from a specified position within a string. The MID$ function takes three arguments, separated by commas: the string from which the characters are to be taken, the position within the string of the first character, and the number of characters desired. For example,

```
MID$ ("HOW DO I LOVE THEE?", 10, 4)
```
—4 characters beginning at position 10 in string "HOW DO I LOVE THEE?"; yields "LOVE"

```
MID$ (H9$, R + 7, 2 * V)
```
—2 * V characters beginning at position R + 7 in the contents of string variable H9$

Third argument optional

You may optionally leave out the third argument to MID$. If you don't specify the number of characters you want, or if the number of characters you request is greater than the length of the string, MID$ yields all characters from the designated starting position to the end of the string:

```
MID$ ("THERE THEY GO!", 7)
```
— all characters from position 7 to end of string "THERE THEY GO!"; yields "THEY GO!"

```
MID$ (A$, 10)
```
— all characters from position 10 to end of the contents of string variable A$

```
MID$ ("HI THERE", 4, 20)
```
— all characters from position 4 to end of string "HI THERE"; yields "THERE"

Real arguments converted to integers

If the value you give for the starting position or the number of characters in the substring is a real number, MID$ truncates it to the next lowest integer. If the designated starting point is greater than the length of the string, or if the number of characters requested is 0, MID$ yields the null string.

null string: a string containing no characters

The starting position must be between 1 and 255, and the number of characters between 0 and 255, or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

If you omit the dollar sign ($) from the function name MID$, Applesoft will treat MID as an arithmetic variable name, causing an error stop with the message

```
?TYPE MISMATCH ERROR
```

The RIGHT$ *Function*

RIGHT$ **extracts a substring from the end of a string**

RIGHT$ extracts a specified number of characters from the end (right end) of a string. The RIGHT$ function takes two arguments, separated by a comma: the string from which the characters are to be taken and the number of characters desired. For example,

```
RIGHT$ ("GIMME A BREAK", 7)
```

—last 7 characters of the string "GIMME A BREAK"; yields "A BREAK"

```
RIGHT$ (NAME$, C + 2)
```

— last C + 2 characters of the contents of string variable NAME$

Real arguments converted to integers

If the value you give for the number of characters in the substring is a real number, RIGHT$ truncates it to the next lowest integer. If the value specified is greater than the length of the string, Applesoft returns the entire string; no extra characters are added.

The number of characters requested must be between 1 and 255 or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

If you omit the dollar sign ($) from the function name RIGHT$, Applesoft will treat RIGHT as an arithmetic variable name, causing an error stop with the message

```
?TYPE MISMATCH ERROR
```

4.2.5 *String Conversion Functions*

Strings and numbers are not the same, even when the string looks like a number:

```
2 * 123                  — yields 246
2 * "123"                — TYPE MISMATCH error
LEFT$ ("123", 2)         —yields "12"
LEFT$ (123, 2)           —TYPE MISMATCH error
```

This section describes Applesoft's built-in functions for converting between numeric and string values:

- STR$ converts a number to a corresponding string.
- VAL converts a string to a corresponding number.
- CHR$ converts an ASCII code to the corresponding character.
- ASC converts a character to the corresponding ASCII code.

ASCII code: see Section 4.2.1

The STR$ *Function*

STR$ converts a number to a string

The STR$ (for "string") function converts a numeric value into a string representing that value. For example,

```
STR$ (-100)              — a string representing the number -100; yields "-100"
STR$ (3.14159)           — a string representing the number 3.14159; yields "3.14159"
STR$ (MARK)              — a string representing the numeric value of real variable MARK
STR$ (COUNT%)            — a string representing the numeric value of integer variable COUNT%
STR$ (B^2 - 4*A*C)       — a string representing the numeric value of the expression B^2 - 4*A*C
```

The string produced by STR$ is in the same format that Applesoft uses to display or print numbers; see Appendix I, "Display Formats for Numbers," for details. For example,

```
STR$ (100 000 000)      — yields "100000000"
STR$ (1 000 000 000)    — yields "1E+09"
STR$ (-.03)             — yields "-.03"
STR$ (-.003)            — yields "-3E-03"
```

If the numeric value of the argument falls outside the allowable range for real numbers (-9.99999999E+37 to +9.99999999E+37), the program will halt with the message

```
?OVERFLOW ERROR
```

The VAL *Function*

VAL **converts a string to a number**

The VAL (for "value") function converts a string to the numeric value it represents. For example,

```
VAL ("4096")                — number represented by the
                              string "4096"; yields 4096

VAL ("-1.505E+2")           — number represented by the
                              string "-1.505E+2";
                              yields -150.5

VAL (WHOLE$ + "." + FRAC$)
                            — number represented by the
                              concatenation of strings
                              WHOLE$, ".", and FRAC$

VAL ( RIGHT$ (Q$, 4) )
                            — number represented by the
                              last 4 characters of string Q$
```

RIGHT$ **function:** see Section 4.2.4

VAL recognizes same number formats as INPUT; see Section 5.1.2

VAL recognizes the same number formats that can be used in keyboard input; see "Rules for Numeric Input" in Section 5.1.2, "The INPUT Statement." If VAL encounters a non-numeric character in its argument string, it yields the numeric value of everything up to the first non-numeric character, ignoring the rest of the string. (The digits 0 through 9, the signs + and -, the decimal point (.), and the letter E for scientific notation are considered numeric characters. Spaces are also allowed, and are simply ignored.) If the first character in the string is non-numeric, VAL yields a value of 0. For example,

```
VAL ("12.54 OR 50")     —yields 12.54
VAL ("ABOUT 4.57")      —yields 0
```

If the absolute value of VAL's result is greater than 1E38 or contains more than 38 digits (including trailing zeros), the program will halt with the message

```
?OVERFLOW ERROR
```

The CHR$ *Function*

CHR$ converts an ASCII code to the corresponding character

The CHR$ (for "character") function regards its single numeric argument as an ASCII character code and yields a one-character string consisting of the corresponding character. For example,

ASCII code: see Section 4.2.1 and Appendix C

```
CHR$ (68)          —character with ASCII code 68; yields the string "D"
CHR$ (47)          —character with ASCII code 47; yields the string "/"
CHR$ (7)           —character with ASCII code 7; yields a string containing the ASCII bell character (CONTROL-G)
CHR$ (C1)          —character whose ASCII code is the value of variable C1
CHR$ (L% + 64)     —character whose ASCII code is the value of expression L% + 64
```

Real arguments converted to integers

If the value of the argument is a real number, CHR$ truncates it to the next lowest integer. For example,

```
CHR$ (81.9)        —argument truncated to 81; yields "Q"
```

An argument less than 0 or greater than 255 will cause the program to halt with the message

```
?ILLEGAL QUANTITY ERROR
```

The ASC *Function*

ASC converts a character to the corresponding ASCII code

The ASC (for "ASCII") function takes a single string argument and yields the ASCII code corresponding to the first character in the string. For example,

ASCII code: see Section 4.2.1 and Appendix C

```
ASC ("D")
```
—ASCII code for character D; yields 68

```
ASC ("/")
```
—ASCII code for character /; yields 47

```
ASC ("e. e. cummings")
```
—ASCII code for character e; yields 101

```
ASC (B0$)
```
—ASCII code for the first character in string B0$

MID$ function: see Section 4.2.4

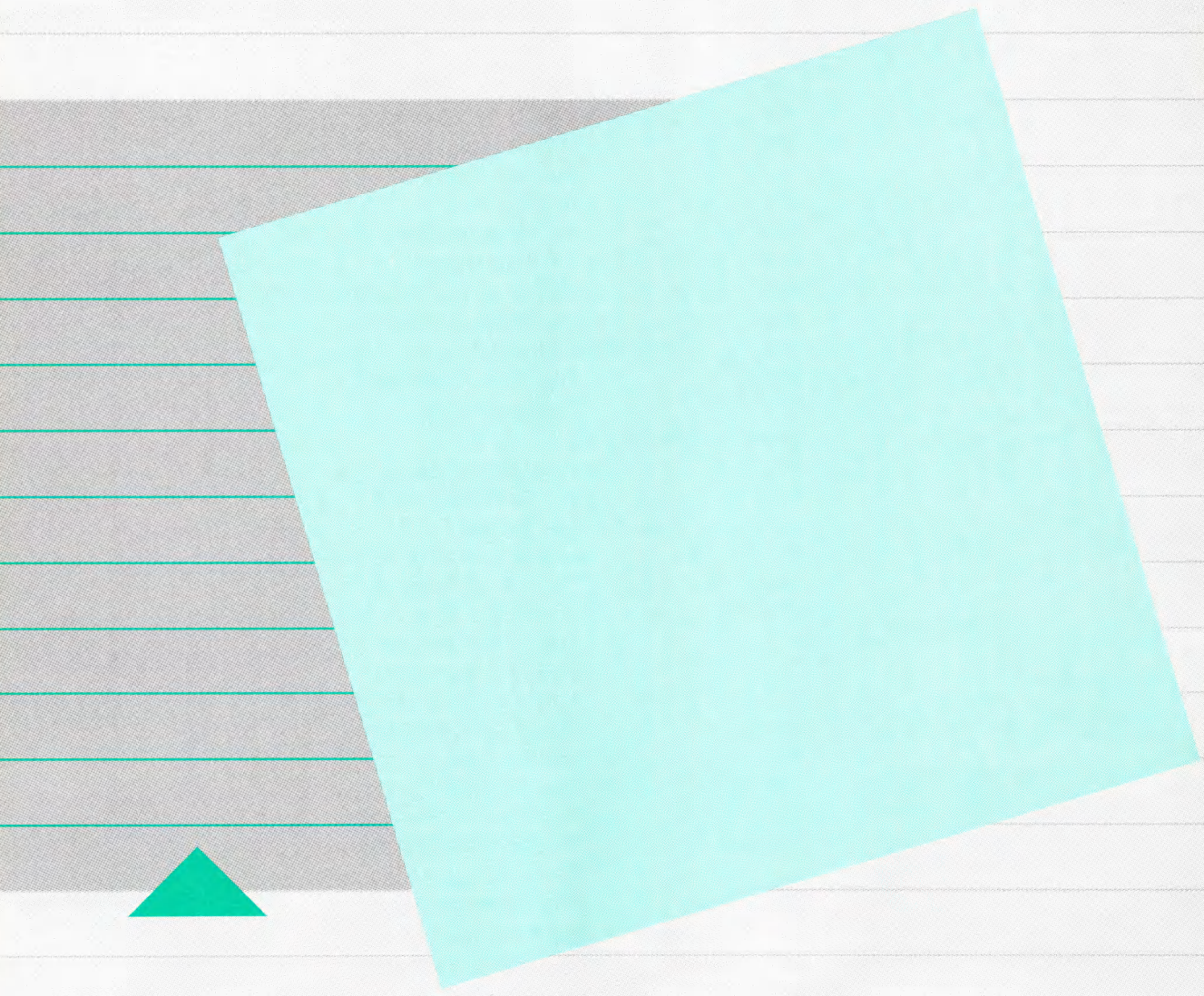
```
ASC ( MID$ (NAME$, 5) )
```
—ASCII code for the fifth character in string NAME$

null string: a string containing no characters

If the argument given to ASC is the null string, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

Input/Output



Input/Output

This chapter is concerned with the ways in which Applesoft programs communicate with the outside world. Here are described Applesoft's facilities for getting information into and out of the computer and for controlling the way information is presented.

input: see Section 5.1

Section 5.1, "Input," deals with the various statements through which Applesoft programs receive information for processing.

output: see Section 5.2

Section 5.2, "Output," describes how programs transfer information to the "outside world": to the display screen, printers, and so forth.

Input

5.1

input: the transfer of information into the computer from an external source

The *input statements* discussed in this section enable Applesoft programs to receive information for processing, either from the keyboard or from a peripheral device connected to the computer via one of the expansion slots:

`IN#` **statement:** see Section 5.1.1

- The `IN#` statement controls the source from which the computer receives its input.

`INPUT` **statement:** see Section 5.1.2

- The `INPUT` statement accepts a line of input from the current input device.

`GET` **statement:** see Section 5.1.3

- The `GET` statement reads a single character from the current input device.

`READ` **statement:** see Section 5.1.4

`DATA` **statement:** see Section 5.1.4

`RESTORE` **statement:** see Section 5.1.5

- The `READ`, `DATA`, and `RESTORE` statements are used to read information from within the running program itself.

miscellaneous input: see Section 5.1.6

- A few miscellaneous input facilities are available for reading the hand controls and for reading information from a cassette tape recorder.

5.1.1 *The* IN# *Statement*

```
IN# 2
IN# X
IN# SLOT - J
```

IN# specifies source for subsequent input

The IN# statement specifies the source from which the computer will receive subsequent input. The expression following the keyword IN# should evaluate to a number between 0 and 7, designating the expansion slot from which input is to be taken.

expansion slot: see *Apple IIe Owner's Manual* and *Apple IIe Reference Manual*

When Applesoft is started up, it is set to receive input from the keyboard. Executing an IN# statement with a slot number from 1 to 7 instructs Applesoft to receive input instead from the peripheral input device (such as a terminal or modem) connected to the designated slot. A slot number of 0 reestablishes the keyboard as the current input device. For example, the following program fragment reads a single character from the device connected to slot 2, then reestablishes keyboard input:

Slot number 0 specifies input from keyboard

```
510 IN# 2          —accept input from device in slot 2
520 GET A$         —read one character from device in slot 2
530 IN# 0          —accept future input from keyboard
```

GET **statement:** see Section 5.1.3

Notice that the character # is part of the keyword IN# and cannot be omitted.

Restarting the System with IN#: If the slot designated in an IN# or PR# statement contains a disk controller card, Applesoft will attempt to restart (often called "booting") the system from the disk contained in drive 1 connected to that slot. When you do this on purpose, it's the usual way of restarting the system from within Applesoft; when you do it by mistake, it can be a catastrophe.

PR# **statement:** see Section 5.2.1

Be careful!

Warning

If no input device is connected to the slot designated in an IN# statement, the system will hang. To recover, use CONTROL-RESET.

CONTROL-RESET **:** see Section 1.3.2

A slot number between 8 and 255 will cause unpredictable and possibly aberrant behavior.

A slot number less than 0 or greater than 255 will stop the program with the message

```
?ILLEGAL QUANTITY ERROR
```

5.1.2 *The* INPUT *Statement*

```
INPUT PRICE
INPUT MNTH%, DAY%, YEAR%
INPUT "WHAT IS YOUR PASSWORD? ";
  PASSWD$
INPUT ""; X
```

INPUT reads a line of input

current input device: see Section 5.1.1

The INPUT statement accepts a line of input (terminated by RETURN) from the current input device, containing values to be assigned to one or more variables. The variables to be read are listed in the INPUT statement, separated by commas.

prompt: to remind or signal the user that some action is expected

current output device: see Section 5.2.1

Prompting message optional

The INPUT statement may optionally include a message to be displayed or printed on the current output device, *prompting* the user for the desired input. If present, the prompting message must be given as a string constant immediately following the keyword INPUT and followed by a semicolon to separate it from the list of variable names. The specified prompting string is reproduced exactly as written; if displayed on the screen, it is immediately followed on the same line by the cursor. If the prompting message is omitted from the INPUT statement, a question mark (?) is used; the question mark can be suppressed by supplying a null string as the prompting message. For example,

```
10 PRINT "WHAT IS YOUR AGE, PLEASE?"
                          —display prompting message on
                            its own line
20 INPUT AGE              —prompt with ? and wait for
                            response
30 INPUT "YOUR STREET NAME? "; ST$
                          —display prompting message on
                            same line as cursor and wait
                            for response
40 PRINT "PLEASE TYPE YOUR FIRST AND
     LAST NAMES, SEPARATED BY A COMMA:"
                          —display prompting message on
                            its own line
50 INPUT ""; FN$, LN$     —suppress ? and wait for two
                            responses separated by a
                            comma
```

The INPUT statement in line 20 above displays a question mark to prompt the user for input, followed by the cursor. The INPUT statement in line 30 displays the prompting message YOUR STREET NAME? instead of the question mark, again followed by the cursor. The INPUT statement in line 50 displays the cursor only, with no question mark and no prompting message of any kind.

Colon causes remainder of line to be ignored

If the user types a colon (:) as part of an input line, the remainder of that input line is ignored. The ASCII null character (CONTROL-@) has the same effect.

CONTROL-C: see Section 1.3.2

An INPUT statement can be interrupted by CONTROL-C, but only if it is the first character typed on an input line. The program halts when the RETURN key is pressed at the end of that line. A CONTROL-C that is not the first character of the input line is treated as part of the input, the same as any other character.

Length of input line limited

Be sure to give your users clear instructions about how long their responses can be. If the user types an input line longer than 255 characters, the whole line will be canceled and will have to be retyped from the beginning (the Apple IIe's speaker will beep from about the 245th character, but no message will be displayed). A response of more than 239 but fewer than 255 characters will be truncated to 239 characters with no warning message displayed.

No INPUT in immediate execution

The INPUT statement can be executed only from within a program; you can't use this statement in immediate execution.

Multiple Inputs on the Same Line

The INPUT statement may list any number of variables to be read from the same input line. The user's responses to these variables must be separated by commas. You can mix string and numeric variables in the same INPUT statement, but the user's responses must each be of the correct type.

If the user presses the RETURN key (or types a colon or CONTROL-@) without typing enough responses for all the variables listed in the INPUT statement, Applesoft displays two question marks to show that it expects a further response. If a colon, comma, or CONTROL-@ is the first character of a response, Applesoft interprets the response as zero or as the null string (depending on the type of variable specified) and the program continues with the next statement.

If the user types more responses than Applesoft expects, or types a colon into the final expected response, Applesoft displays the message

```
?EXTRA IGNORED
```

and program execution continues. If the last response is shortened by a CONTROL-@—the program continues but no message is displayed.

Programming Tip: Multiple inputs on the same line can be confusing for your users; it's best not to use them except for "quick and dirty" testing purposes while you're debugging your code. Instead of asking for something terribly unfriendly like

```
PLEASE TYPE LAST NAME, FIRST NAME, MIDDLE
INITIAL:
```

Be kind to your users!

use a form such as

```
PLEASE TYPE YOUR FIRST NAME:
```

followed by

```
PLEASE TYPE YOUR MIDDLE INITIAL;
JUST PRESS RETURN IF YOU HAVE NONE
```

and so on. You'll be able to give much clearer instructions, your user will have an easier time giving you what you want, and you'll be better able to detect and deal with errors in the input.

Rules for String Input

The following rules govern the responses the user types to string variables in the INPUT statement:

Quotation marks optional

- The user's response to a string variable may be typed with or without enclosing quotation marks.

Leading spaces ignored

- Applesoft ignores all spaces preceding the first nonspace character.

Rules for **quoted responses**

- If the first nonspace character is a quotation mark, the input string is considered to include everything up to (but not including) the next quotation mark, CONTROL-@, or RETURN. The string may include commas and colons, but may not include quotation marks, since these would be interpreted as marking the end of the string. Spaces following the closing quotation mark are ignored, but any other character causes the response to be rejected with the message

  ```
  ?REENTER
  ```

Rules for **unquoted responses**

- If the first nonspace character is not a quotation mark, the input string includes everything up to (but not including) the next comma, colon, CONTROL-@, or RETURN. The string may include quotation marks, but may not include commas or colons, since these would be interpreted as marking the end of the string. Spaces following the last nonspace character are accepted as part of the input string.

Null responses OK

- If the first nonspace character is a comma, colon, CONTROL-@, or RETURN, the response is interpreted as the null string and program execution continues.
- The following control characters cannot be included in the response:
  - CONTROL-H (equivalent to the LEFT-ARROW or backspace key)
  - CONTROL-M (equivalent to the RETURN key)
  - CONTROL-X (cancels the input line)
  - CONTROL-@ (ASCII null character; causes remainder of input line to be ignored)

Control characters cause problems

In general, control characters cause problems and should not be used in responding to INPUT statements.

String expressions don't work

- The response to a string variable must be a single string or a constant; it cannot be a string expression involving concatenation, LEFT$, MID$, RIGHT$, or other string operations. Responses such as

```
A$ + B$
LEFT$ (MNTH$, 3)
RIGHT$ (NAME$, L - (FL + 2))
```

will be accepted exactly as typed, character for character (up to the first comma), and will not be evaluated as string expressions.

Rules for Numeric Input

Listed below are the rules governing the user's responses to numeric variables. If a response is typed that doesn't conform to these rules, Applesoft will display the message

```
?REENTER
```

reissue the prompting message, and wait for another response.

All **spaces** ignored

- Spaces are ignored in any position.

- The response is considered to include all nonspace characters up to (but not including) the next comma, colon, CONTROL-@, or RETURN.

Numeric characters only

scientific notation: see Section I.2

- The response may include numeric characters and spaces only. Numeric characters include the digits 0 to 9, the signs + and -, the period (decimal point), and the letter E for scientific notation. A response containing a non-numeric character in any position is invalid.

Form of numbers

- Numeric responses consist of the following elements. Any or all of these elements may be omitted, except that the sign or value of the exponent may not appear unless preceded by the letter E. Those that are included must be given in the order listed:
  - A sign (+ or −)
  - One or more digits
  - A decimal point (.)
  - One or more digits
  - The letter E for scientific notation
  - A sign (+ or −) for the exponent
  - One or more digits

Degenerate cases interpreted as 0

Even forms such as +E- and .E are accepted, and are interpreted as 0.

Null responses interpreted as 0

- If the first nonspace character is a comma, colon, or RETURN, the response is interpreted as 0 and program execution continues. A response beginning with CONTROL-@ is invalid.
- The following control characters have special meanings:
  - CONTROL-H (equivalent to the LEFT-ARROW or backspace key)
  - CONTROL-M (equivalent to the RETURN key)
  - CONTROL-X (cancels the input line)
  - CONTROL-@ (ASCII null character; causes remainder of input line to be ignored)

Most **control characters** illegal

A response containing any other control character, in any position, is invalid.

Arithmetic expressions invalid

- The response to a numeric variable must be a single number; it cannot be a numeric expression involving arithmetic operations

or function calls. Responses such as

```
1/2
B^2 - 4*A*C
SQR (2)
```

are invalid because of the non-numeric characters.

VAL **function:** see Section 4.2.5

It's a good idea to use string variables to accept all numeric inputs, using the VAL function to convert them to numeric values. This makes it easier to detect and deal with user errors and to display alternate prompting messages.

An "Input Anything" Routine

The INPUT statement interprets the colon and the comma (and sometimes the quotation mark) as special symbols and rejects anything typed after them in the input line. Here's a bit of magic you can use if you anticipate that your user's response may include any of these characters.

POKE **statement:** see Section 7.1.2

The following Applesoft subroutine uses the POKE statement to store a special machine-language routine into the computer's memory, one byte at a time, beginning at address 768. The machine-language routine will accept all characters in the input, including colons, commas, and quotation marks, without "censoring" them, and will assign them, character by character, to a string variable for further processing. (The line numbers used below are arbitrary; you can locate this subroutine anywhere you like in your program.)

```
62000 REM SET UP "INPUT ANYTHING"
        ROUTINE
62010 LET IN$ = "X"     —IN$ must be first variable
                          created
62020 FOR J = 768 TO 790
                        —these are memory addresses
                          where machine language is to
                          be stored
62030 READ I            —get a byte of machine
                          language
62040 POKE J, I         —store it at next location
62050 NEXT J            — go back for next byte
62060 DATA 162, 0, 32, 117, 253, 160, 2,
        138, 145, 105, 200, 169, 0, 145,
        105, 200, 169, 2, 145, 105, 76,
        57, 213         —these are the actual bytes of
                          machine language
62070 RETURN            —return to statement following
                          point of call
```

The DATA statement containing the machine language must be reproduced in your program exactly as shown.

CALL **statement:** see Section 7.1.3

The following subroutine uses the CALL statement to call the machine-language routine at address 768. (Again, this subroutine can be located anywhere in your Applesoft program, not necessarily at line number 63000.)

```
63000 REM CALL "INPUT ANYTHING" ROUTINE
63010 CALL 768          —call machine-language routine
63020 IN$ = MID$ (IN$, 1)
                        —IN$ now holds the input that
                         the machine-language routine
                         accepted
63030 RETURN            —return to statement following
                         point of call
```

To accept a line of input from the user, instead of using a statement such as

```
100 INPUT S$
```

substitute this line:

```
100 GOSUB 63000 : LET S$ = IN$
```

The variable S$ now contains whatever input the user typed, including the "forbidden" characters; your program can proceed to process the input in whatever way is appropriate.

For technical reasons having to do with the way variables are stored in memory, the string variable used to pass the user's response between machine language and Applesoft (arbitrarily called IN$ in the example above) must be the first variable used or defined in the program. To be safe, you might want to call the "input anything" setup routine from line number 0:

```
0 GOSUB 62000
```

5.1.3 *The* GET *Statement*

```
GET L$
GET S$(N)
GET C1$, C2$, C3$
```

GET reads a single input character

current input device: see Section 5.1.1

The GET statement reads a single character from the current input device. Although it can be used to read from any peripheral input device (such as a terminal or modem), it is seldom used in actual practice with anything other than the keyboard.

GET accepts one character from the current input device for each of the string variables listed following the keyword GET. Each single character is read as soon as it is typed, without waiting for the user to press the RETURN key. The character is not displayed on the screen, and the cursor is not moved in any way.

Here's an example of a program fragment using GET:

```
310 PRINT "PRESS THE 'Y' KEY TO GO ON:";
                          —prompt user for response
                           (semicolon keeps cursor on
                           same line)
320 GET A$                —wait for user to press key
330 IF A$ <> "Y" THEN 320
                          —keep cycling until user presses
                           correct key
340 PRINT                 —move cursor to new line (can-
                           cels effect of semicolon from
                           line 310)
350 PRINT "THANK YOU"     —politeness from machines is
                           always welcome
```

semicolon: see Section 5.2.2

No GET in immediate execution

The GET statement can be executed only from within a program; you can't use this statement in immediate execution.

CONTROL-C won't interrupt a GET

CONTROL-C: see Section 1.3.2

If typed in response to a GET, CONTROL-C is treated like any other character; it does not interrupt program execution.

DOS: Disk Operating System

A DOS command issued immediately after a GET will not be recognized. For DOS commands to be executed properly, you must issue a RETURN character immediately after the GET and before the DOS command. An easy way to do this is with an empty PRINT statement:

```
PRINT
```

See your DOS manual for more information.

Numeric Inputs with GET: The GET statement is neither designed nor intended to obtain values for numeric variables. You may attempt to do so at your own peril, subject to the following limitations:

- A comma or a colon will result in the message

  ```
  ?EXTRA IGNORED
  ```

  and will be interpreted as a numeric value of 0.
- A plus sign, minus sign, CONTROL-@, E, space, or period will be interpreted as a numeric value of 0.
- Any non-numeric character will cause the program to halt with a syntax error.

It's better to use only string variables with the GET statement, using the VAL function to convert the response to a numeric value.

VAL **function:** see Section 4.2.5

5.1.4 *The* READ *and* DATA *Statements*

```
READ PRICE
READ A, B, M%(I), J%, S$(2*J - 1), T$
DATA 12.9, HI HO, 168
DATA 2.236
```

READ **reads information from body of program**

DATA **sets up information for use by** READ

The READ and DATA statements are used to read information from within the body of the Applesoft program itself, rather than from the keyboard or an input device. There may be any number of DATA statements in a program, each containing a list of one or more items of information (numbers or strings) following the keyword DATA and separated by commas. All of the DATA statements in the program are considered to form one long list of items, in sequential order of line numbers; each READ statement reads one or more items from this list.

Each time it executes a READ statement, Applesoft remembers the last item read from the DATA list. The next READ always begins with the next item in the list. There is no way to "back up" or "skip forward" in the DATA list, but you can start over from the beginning of the list with the RESTORE statement.

RESTORE **statement:** see Section 5.1.5

Don't read past end of list!

An attempt to read past the end of the DATA list will halt the program with a message such as

```
?OUT OF DATA ERROR IN 1465
```

identifying the line number of the READ statement in which the error occurred. Leaving part of the DATA list unread at the end of the program does not cause an error.

Rules for numeric and string input: see Section 5.1.2

The items in a DATA statement are separated by commas and follow the usual rules for numeric and string input, except that a DATA statement cannot contain a colon (:). The number of items in each DATA statement is limited only by the length of the program line. A DATA statement may appear anywhere in your program; it need not precede the READ statement that uses it. There is no limit to the number of DATA statements in a program.

DATA statements may appear anywhere

Here's an example program showing the use of the READ and DATA statements:

```
10 DATA "GO WEST, YOUNG MAN"
```
—item containing a comma; OK between quotation marks

```
20 DATA 3.14159, 2, "SAM"
```
—mixed types in same DATA statement

```
30 READ A$, B
```
—read GO WEST, YOUNG MAN into string variable A$ and 3.14159 into real variable B; notice that these items come from two different DATA statements

```
40 READ C%, D$, E$
```
—read 2 into integer variable C%, SAM into string variable D$, and THE "WORLD" IS FLAT into string variable E$; begins with next item following previous READ statement

```
50 DATA THE "WORLD" IS FLAT
                          —item containing quotation
                           marks; notice that this item fol-
                           lows the READ statement that
                           uses it
60 PRINT E$               —display THE "WORLD"
                           IS FLAT
70 PRINT A$               —display GO WEST,
                           YOUNG MAN
80 DATA 98.6, 37, -273.16
                          —these items never read
90 END
```

Null items interpreted as 0 or null string

Null items in a DATA statement are interpreted as 0 or the null string, depending on the type of variable to which they are assigned in a READ statement. A null item is read whenever there are no non-space characters

- between the keyword DATA and the end of the program line
- between the keyword DATA and the first comma
- between two consecutive commas
- between the last comma and the end of the program line

Thus the statement

```
DATA,,
```

contains three null items.

VAL **function:** see Section 4.2.5

An attempt to read a string value in a DATA statement with a numeric variable in a READ statement causes a syntax error. Numeric values can be read into string variables, but must be evaluated with the VAL function before they can be used as numbers.

Most **control characters** treated as ordinary characters

CONTROL-C: see Section 1.3.2

The characters CONTROL-H, CONTROL-M, CONTROL-X, and CONTROL-@ cannot be embedded in a DATA statement. Any other control character typed into a DATA statement is treated as an ordinary character and becomes part of the input. A CONTROL-C character in a DATA statement will not interrupt the program.

The READ statement can be executed only from within a program; you can't use this statement in immediate execution.

5.1.5 *The* RESTORE *Statement*

```
RESTORE
```

RESTORE **restarts** DATA **list**

The RESTORE statement restarts the DATA list from the beginning. After RESTORE is executed, the next READ statement will read the first item in the first DATA statement in the program. For example,

```
 10 DATA "GO WEST, YOUNG MAN"
 20 DATA 3.14159, 2, "SAM"
 30 READ A$, B          —read GO WEST, YOUNG
                          MAN into string variable A$
                          and 3.14159 into real vari-
                          able B
 40 READ C%, D$, E$     —read 2 into integer variable C%,
                          SAM into string variable D$,
                          and THE "WORLD" IS
                          FLAT into string variable E$
 50 DATA THE "WORLD" IS FLAT
 60 PRINT E$            —display THE "WORLD"
                          IS FLAT
 70 RESTORE             —restart list from beginning
 80 READ Q$             —read GO WEST, YOUNG
                          MAN into string variable Q$
 90 PRINT Q$            —display GO WEST,
                          YOUNG MAN
100 PRINT A$            —display GO WEST,
                          YOUNG MAN (value of A$
                          still intact)
110 END
120 DATA 98.6, 37, -273.16
                        —these items never read
```

There is no easy way to reposition the DATA list to a specific desired item or line number. The only other Applesoft statement that affects the positioning of the DATA list is RUN, which also restarts the list from the beginning.

5.1.6 *Miscellaneous Input Facilities*

This section covers Applesoft's facilities for dealing with the remaining input features of the Apple IIe: the hand controls and cassette tape input.

The Hand Controls

PDL reads dials on hand controls

If you have a set of hand controls connected to your computer, you can use the PDL function to read their dial settings. The Apple IIe can accommodate as many as four hand controls, numbered 0 to 3, connected through the 9-pin hand control connector on the computer's back panel or the GAME I/O connector inside the case on the main logic board. However, the standard Apple hand control set consists of only two controls, numbered 0 and 1.

Standard hand controls numbered 0 and 1

Result of PDL is between 0 and 255

The PDL function takes one argument, the number of the hand control to be read, and yields an integer from 0 to 255 representing the current position of the dial on that control. For example,

```
10 LET X = PDL (0)    —read hand control 0
20 LET P% = X * 40 / 256+1
                      —reduce to a number from 1 to
                        40
30 HTAB P%            —move cursor to indicated posi-
                        tion on current line
40 PRINT "<"          —display the character <
50 LET Y = PDL (1)    —read hand control 1
60 LET Q% = Y * 40 / 256+1
                      —reduce to a number from 1 to
                        40
70 HTAB Q%            —move cursor to indicated posi-
                        tion on current line
80 PRINT ">"          —display the character >
90 IF X = 0 AND Y = 0 THEN END
                      —end program when both hand
                        controls read 0
100 GOTO 10           —otherwise repeat the process
```

If the argument given to PDL is less than 0 or greater than 255, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

If the argument is between 4 and 255, or if no hand control of the designated number is connected, the results are unpredictable.

Allow a delay between calls to PDL

If your program reads two hand controls in consecutive statements, the reading from the first hand control may affect the reading from the second. To obtain more accurate readings, allow several program lines between calls to PDL or use a short delay loop such as

```
FOR X = 1 TO 10 : NEXT X
```

between PDL calls.

Historical Note: The function name PDL stands for "paddle," which in turn is short for "game paddle," an older name for the Apple IIe's hand controls.

Reading the **hand control buttons**

The buttons on the hand controls can be read with the function calls

PEEK **function:** see Section 7.1.1

`PEEK (-16287)`	—yields a value > 127 if button on hand control 0 is being pressed, <= 127 if not
`PEEK (-16286)`	—yields a value > 127 if button on hand control 1 is being pressed, <= 127 if not
`PEEK (-16285)`	—yields a value > 127 if button on hand control 2 is being pressed, <= 127 if not

There is no way to read the button on hand control 3. The PEEK calls listed above are also used to read the "apple keys" on the Apple IIe keyboard: the OPEN-APPLE key is equivalent to the button on hand control 0, and SOLID-APPLE is equivalent to the button on hand control 1.

For more information...

See the Apple IIe Reference Manual for detailed technical information on the 9-pin hand control connector and the internal GAME I/O connector.

Cassette Input

LOAD **command:** see Section 1.2.6 and Appendix M

RECALL **statement:** see Appendix M

SHLOAD **statement:** see Section 6.3.2 and Appendix M

Three Applesoft statements, LOAD, RECALL, and SHLOAD, can be used to read information from a cassette tape recorder. LOAD reads an Applesoft program into memory from tape; RECALL reads the contents of an integer or real array; SHLOAD reads a shape table for use in high-resolution graphics. For details, see Appendix M, "If You Have a Cassette Recorder."

5.2 Output

output: the transfer of information from the computer to an external destination

This section describes the *output* facilities available in Applesoft:

PR# statement: see Section 5.2.1

- Section 5.2.1 covers the PR# statement, which controls the destination to which output is directed.

PRINT statement: see Section 5.2.2

- Section 5.2.2 contains a detailed discussion of the PRINT statement, Applesoft's primary output statement.

number formats: see Section 5.2.3

- Section 5.2.3 gives details on the way numbers are formatted when written with the PRINT statement.

screen formatting: see Section 5.2.4

- Section 5.2.4 describes Applesoft's wide variety of facilities for controlling the format in which textual information is displayed on the screen.

miscellaneous output: see Section 5.2.5

- Section 5.2.5 touches briefly on various miscellaneous output facilities not covered elsewhere: the Apple IIe's built-in speaker, annunciator outputs, utility strobe, and cassette tape output.

5.2.1 The PR# Statement

```
PR# 1
PR# X
PR# SLOT - J
```

PR# specifies destination for subsequent output

The PR# statement specifies the destination to which the computer will send subsequent output. The expression following the keyword PR# should evaluate to a number between 0 and 7, designating the expansion slot to which output is to be sent.

expansion slot: see *Apple IIe Owner's Manual* and *Apple IIe Reference Manual*

When Applesoft is started up, it is set to send output to the display screen. Executing a PR# statement with a slot number from 1 to 7 instructs Applesoft to send output instead to the peripheral output device (such as a printer, terminal, or modem) connected to the designated slot. A slot number of 0 reestablishes the display screen as the current output device. For example, the following program fragment writes a string of characters to the device connected to slot 1, then reestablishes screen output:

Slot number 0 specifies output to the screen

```
610 PR# 1          —send output to device in slot 1
620 PRINT Z$       —write contents of string variable Z$ to device in slot 1
630 PR# 0          —send future output to screen
```

Notice that the character # is part of the keyword PR# and cannot be omitted.

IN# **statement:** see Section 5.1.1

Be careful!

Restarting the System with PR#: If the slot designated in an IN# or PR# statement contains a disk controller card, Applesoft will attempt to restart (often called "booting") the system from the disk contained in drive 1 connected to that slot. When you do this on purpose, it's the usual way of restarting the system from within Applesoft; when you do it by mistake, it can be a catastrophe.

Warning

If no output device is connected to the slot designated in a PR# statement, the system will hang. To recover, use CONTROL - RESET .

CONTROL - RESET : see Section 1.3.2

A slot number between 8 and 255 will cause unpredictable and possibly aberrant behavior.

Warning

If you are using the Apple IIe 80-Column Text Card, always be sure to deactivate it by typing ESC CONTROL -Q before using PR# to transfer output to another slot. Leaving the Text Card active while using a printer or while restarting the system from a disk may produce amusing but confusing fireworks on the screen.

Although the Text Card is installed in the Apple IIe's special auxiliary slot, it appears to the computer as if it were in slot 3. So to reactivate the Text Card after sending output to another device, type

```
PR# 3
```

You can also return output to the 40-column screen with the Text Card inactive by typing

```
PR# 0
```

However, don't use PR# 0 to redirect output directly from the Text Card to the 40-column screen without first deactivating the Text Card with ESC CONTROL -Q. Under certain circumstances, this may cause text intended for the screen to be written outside the area of memory reserved for it, possibly destroying your Applesoft program or other important information.

A slot number less than 0 or greater than 255 will stop the program with the message

```
?ILLEGAL QUANTITY ERROR
```

5.2.2 *The* PRINT *Statement*

```
PRINT
PRINT P$, Q, R%
PRINT "DISCRIMINANT = "; B^2 - 4*A*C
PRINT LEFT$ (FN$, 1) + "," + LN$
PRINT TAB (M); "*"; TAB (M + N);
  "***"; TAB (M + N + N); "*"
```

PRINT writes to the current output device

The PRINT statement writes output to the current output device. Expressions representing the values to be written are listed after the keyword PRINT, separated by commas or semicolons.

current output device: see Section 5.2.1

Any expression may be included in a PRINT statement. Each expression in the list following the keyword PRINT is evaluated. If the value of the expression is a string, the characters of the string are written to the current output device; if the value is a number, it is written according to the rules discussed in Section 5.2.3, "Number Formats." Calls to the special functions SPC and TAB may also be included in a PRINT list; they do not cause anything to be written, but control the positioning of the next item.

number formats: see Section 5.2.3

SPC **function:** see Section 5.2.4

TAB **function:** see Section 5.2.4

Semicolon suppresses space after an item

When an item in the PRINT list is followed by a semicolon, the cursor (if output is going to the screen) or print head (if to a printer) is left positioned immediately after the last character in the item. The next item written will begin in the next available column, with no intervening spaces. A semicolon at the end of a PRINT statement causes the cursor or print head to be left at the end of that line, and prevents a new line from being started. For example, the statement

```
PRINT 1; 2; 3; 4;
```

will produce the output

```
1234
```

and will leave the cursor or print head positioned in the column immediately following the digit 4. The statement

```
PRINT 1/3; (2 * 4); 51
```

will produce the output

```
.333333333851
```

If two consecutive items in a PRINT list are not separated by either a comma or a semicolon, a semicolon is understood.

80-Column Text Card: see *Apple IIe Owner's Manual, Apple IIe 80-Column Text Card Manual*

The Apple IIe's normal display is 40 columns wide. After Applesoft displays the 40th character on a line, it automatically sends the cursor to the beginning of the next line. The next PRINT statement executed will start another new line, causing an unintended blank line to appear on the screen. This happens even if you have the Apple IIe 80-Column Text Card installed and running in "active 80" mode; Applesoft doesn't know about the 80-column display and will still break each output line after 40 characters.

For example, the statements

```
10 PRINT "THIS MESSAGE HAS PRECISELY 40
     CHARACTERS"
20 PRINT "SO THERE'S A BLANK LINE ON THE
     SCREEN"
```

will display the output

```
THIS MESSAGE HAS PRECISELY 40 CHARACTERS

SO THERE'S A BLANK LINE ON THE SCREEN
```

To eliminate the blank line, add a semicolon to the end of line 10:

```
10 PRINT "THIS MESSAGE HAS PRECISELY 40
     CHARACTERS";
```

Now you'll get this:

```
THIS MESSAGE HAS PRECISELY 40 CHARACTERS
SO THERE'S A BLANK LINE ON THE SCREEN
```

The second line of this message is now a lie.

A statement such as

```
PRINT A$ + B$
```

causes a halt with the message

```
?STRING TOO LONG ERROR
```

concatenation: see Section 4.2.2

if the combined length of the concatenated strings is greater than 255. However, you can print the apparent concatenation regardless of length by using a semicolon:

```
PRINT A$; B$
```

Comma advances to next tab position

When an item in the PRINT list is followed by a comma, the cursor or print head is advanced to the next available *tab position:* column 17 or 33 of the current line or column 1 of the next line. The next item written will begin at the tab position. A series of consecutive commas will advance the cursor or print head a corresponding number of tab positions. A comma at the end of a PRINT statement causes the cursor or print head to be left at the next available tab position, and may prevent a new line from being started. For example, the statement

```
PRINT 1, 2, 3, 4,
```

will produce the output

```
1               2               3
4
```

and will leave the cursor or print head positioned in column 17 of the second line, directly under the digit 2. The statement

```
PRINT 1/3, (2 * 4), 51
```

will produce the output

```
.333333333      8               51
```

80-Column Text Card: see *Apple IIe Owner's Manual, Apple IIe 80-Column Text Card Manual*

If any character appears in columns 24 to 32, or if you have the Apple IIe 80-Column Text Card installed in your computer and running in "active 80" mode, then column 33 is not available as a tab position; a comma after column 17 will cause the next item to start at column 1 of the next line.

text window: see Section 5.2.4, Section F.1, and the *Apple IIe Reference Manual*

If the text window is set to fewer than 33 columns wide, commas in a PRINT statement do not function properly and may cause text to be displayed outside the text window.

A PRINT statement that doesn't end with a comma or semicolon always starts a new line after writing its last item and leaves the cursor or print head positioned in column 1 of the new line. The statement

```
PRINT
```

PRINT **by itself** starts a new line

simply starts a new line. If the cursor or print head was already at the beginning of a line, this statement causes a blank line to be displayed or printed.

Here's an example program using some of the features of PRINT discussed above:

```
10 LET A = 5.35 : LET C$ = "FRED" : LET
     G% = 16          —set up series of variables
20 PRINT "STUFF AND NONSENSE"
                      —display message and start
                        new line
30 PRINT              —display a blank line
40 PRINT "A = ";      —display message without start-
                        ing a new line
50 PRINT A            —display 5.35 on same line as
                        message from program line
                        40; start new line
60 PRINT "G% = "; G%  —display message and value
                        16 on same line; start new
                        line
70 PRINT "C$ = ", C$  —display message, advance to
                        next tab position, and display
                        string FRED; start new line
80 PRINT A * G%       —display value of expression
                        A * G% (85.6) and start
                        new line
```

When executed, this program will produce the following output:

```
STUFF AND NONSENSE

A = 5.35
G% = 16
C$ =                FRED
85.6
```

? stands for PRINT

Abbreviation: You can use a question mark (?) as an abbreviation for the keyword PRINT; if you use it, it appears as PRINT in a program listing. If you type

```
100 ? A$              —display string A$
LIST                  —list program
```

Applesoft will display

```
100 PRINT A$          —Applesoft sees ? as PRINT
```

5.2.3 *Number Formats*

This section describes the formats in which Applesoft displays or prints numeric values. Numbers may not always be formatted in the way you might expect; this is particularly true for numbers more than nine digits long or for exceptionally small numbers.

Ranges of numeric values

Numeric values in Applesoft must be in the range `-1*10^38` to `1*10^38`. Any number whose absolute value is less than approximately `3*10^-39` is converted to zero. True integer values to be assigned to integer variables (such as `A%`) must be in the range `-32767` to `+32767`.

A number typed from the keyboard or a numeric constant used in an Applesoft program may have as many as 38 digits. However, only nine digits are significant, and the last digit is rounded off. An Applesoft statement that you type as

```
PRINT 1.23456787654321
```

—you type this from the keyboard

will display

```
1.23456788
```

—you get this on the screen

on the screen.

All arithmetic done on reals

Integers are always converted to real form before being used in arithmetic calculations, and the results are converted back to integer form when assigned to an integer variable. Conversion from real to integer form is by truncation to the next lowest integer, not by rounding to the nearest integer.

truncate: to convert a real number to the next lower integer

Rules for number formats

Applesoft displays and prints numbers according to the following rules:

- If the number is negative, it is preceded by a minus sign (`-`); if it is zero or positive, no sign is used.
- If the number is an integer with an absolute value from `0` to `999 999 999`, it is formatted as an integer.

- If the number is not an integer and its absolute value is between .01 and 999 999 999.2, it is formatted with a decimal point in the usual way.
- In all other cases, the number is formatted in scientific notation (see below).

Table 5-1 shows examples of the formats used for displaying and printing numbers.

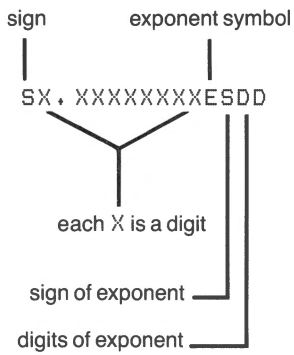
Table 5-1 Number Formats

Number	Output Format
+1	1
−1	−1
6523	6523
−23.460	−23.46
45.72 * 10^5	4572000
1 * 10^20	1E+20
−12.34567896 * 10^10	−1.2345679E+11
1000000000	1E+09
999999999	999999999

scientific notation: the representation of numbers in terms of powers of 10

The format Applesoft uses for scientific notation is shown in Figure 5-1. A sign is shown only if the number is negative. There is always exactly one nonzero digit before the decimal point and up to eight digits after it, with trailing zeros suppressed. There are never any leading zeros; the digit before the decimal point is always nonzero. If there is only one digit to print after all trailing zeros are suppressed, no decimal point is shown. The letter E (for "exponent") is always followed by a sign and a two-digit exponent. The value of a number represented in this form is the number before the E times 10 raised to the power after the E. For example,

Figure 5-1 Format for Scientific Notation




```
PRINT 35 * 345 ^ 14    yields 1.18450085E+37
PRINT -3.14159 * 567 ^ 5
                       yields -1.84104669E+14
PRINT 1 / 999          yields 1.001001E-03
PRINT -3 / 999         yields -3.003003E-03
```

5.2.4 *Formatting Text on the Screen*

This section deals with Applesoft's facilities for controlling the way text is formatted and presented on the display screen. For further information on text formatting, see Section 5.2.2, "The PRINT Statement."

The TEXT and HOME statements are used to clear text and graphics from the screen.

SPC, TAB, HTAB, VTAB, and POS control the position of the cursor, which determines where characters are displayed on the screen.

NORMAL, INVERSE, and FLASH control the form in which text characters are presented on the screen.

The SPEED= command sets the rate at which characters are displayed.

POKE **statement:** see Section 7.1.2

The POKE statement can be used to set the boundaries of the text window within which text is displayed on the screen.

The TEXT *Statement*

```
TEXT
```

TEXT **switches from graphics to text display**

text window: see below

80-Column Text Card: see *Apple IIe Owner's Manual, Apple IIe 80-Column Text Card Manual*

The TEXT statement instructs Applesoft to begin displaying text on the screen; it is usually used to switch from graphics to text display. The text window is set to the full screen (24 lines, 40 characters per line; 80 if the 80-Column Text Card is installed and running in "active 80" mode). The Applesoft prompt character (]) is displayed in the bottom-left corner of the screen, followed by the cursor.

If the display is already in text mode, the TEXT statement is equivalent to the statement VTAB 24 (see "The VTAB Statement," below).

The HOME *Statement*

```
HOME
```

HOME **clears the text window**

text window: see below

HOME clears the currently defined text window and sends the cursor to the top-left corner of the window. If the text window is set to the full screen, the cursor is sent to the beginning of line 1. If the computer is displaying mixed text and graphics (four lines of text at the bottom of the screen), HOME clears the four text lines and sends the cursor to the beginning of line 21.

Helpful Hint: To move the cursor to the top-left corner of the screen without clearing any text, use

```
VTAB 1 : HTAB 1
```

VTAB **and** HTAB **statements:** see below

The SPC *Function*

SPC **displays spaces on the screen**

PRINT **statement:** see Section 5.2.2

The SPC (for "space") function is used in PRINT statements to write a specified number of spaces to the current output device. The numeric argument given to the function specifies the number of spaces to be written. If this value is a real number, Applesoft truncates it to the next lowest integer.

TAB **function:** see below

current output device: see Section 5.2.1

The SPC function can be called only from within a PRINT statement. SPC differs from TAB in that it advances the cursor (or print head, if the current output device is a printer) a specified number of columns from its current position, rather than to a specific horizontal position from the beginning of the current line. If the cursor is spaced past the right edge of the screen or text window, it returns to the beginning of the next line and continues spacing. For example, assuming the text window is set to the full screen and the cursor is initially at the left edge, the statements

```
10 PRINT SPC (5); "HELLO"
                          —display HELLO starting in
                            column 6
20 PRINT "THESE"; SPC (10); "ARE"; SPC
     (4); "INTERESTING"; SPC (12);
     "TIMES"              —display THESE, 10
                            spaces, ARE, 4 spaces,
                            INTERESTING, 12 spaces,
                            TIMES
```

will display the following on the screen:

```
     HELLO
THESE          ARE    INTERESTING
     TIMES
```

Notice how the output of line 20 "wraps around" when it reaches the edge of the screen (column 40).

SPC **at end of** PRINT suppresses new line

If SPC is the last item in a PRINT statement, Applesoft acts as if the statement ended with a semicolon. The cursor is left positioned the specified number of spaces after the end of the previous item; no new line is started. The next item displayed will begin immediately following the last space.

The argument given to SPC must be in the range 0 to 255 or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

However, several calls to SPC can be strung together in the form

```
PRINT SPC(255); SPC(255); SPC(255)
```

to provide arbitrarily large numbers of spaces.

Semicolons are optional between SPC items:

```
PRINT "LET"; SPC (10) "ALL" SPC (15);
  "REJOICE"
```

The TAB *Function*

TAB advances cursor to a specified horizontal position

PRINT statement: see Section 5.2.2

The TAB function is used in PRINT statements to advance the cursor to a specified horizontal position from the beginning of the current output line. The numeric argument given to the function specifies the position to which the cursor is to be moved. If this value is a real number, Applesoft truncates it to the next lowest integer.

SPC function: see above

current output device: see Section 5.2.1

The TAB function can be called only from within a PRINT statement. TAB differs from SPC in that it advances the cursor (or print head, if the current output device is a printer) to a specific horizontal position from the beginning of the current line, rather than a specified number of columns from the current cursor position. If the cursor is advanced past the right edge of the screen or text window, it returns to the beginning of the next line and continues advancing. For example, assuming the text window is set to the full screen and the cursor is initially at the left edge, the statements

```
10 PRINT TAB (15); "THE FLEET'S IN!"
                          —display THE FLEET'S IN!
                            starting at column 15
20 PRINT TAB (10); "HELLO"; TAB (30);
     "THERE,"; TAB (45); "SAILOR!"
                          —display HELLO at column 10,
                            THERE, at column 30,
                            SAILOR! at column 5 of
                            next line
```

will display the following on the screen:

```
              THE FLEET'S IN!
        HELLO                  THERE,
    SAILOR!
```

Notice how the output of line 20 "wraps around" when it reaches the edge of the screen (column 40).

HTAB **statement:** see below

Unlike the HTAB statement, which moves the cursor to an absolute horizontal position from the left edge of the screen or the text window, TAB moves the cursor in a forward direction only. If the specified tab position is less than the current cursor position, TAB has no effect; it will never move the cursor to the left on the current line (use HTAB for this purpose).

TAB **at end of** PRINT suppresses new line

If TAB is the last item in a PRINT statement, Applesoft acts as if the statement ended with a semicolon. The cursor is left at the specified tab position; no new line is started. The next item displayed will begin at the tab position.

The argument given to TAB must be in the range 0 to 255 or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

An argument value of 0 moves the cursor to 256 positions from the beginning of the current line.

Semicolons are optional between TAB items:

```
PRINT "DOWN"; TAB (14) "YOU" TAB (27); "GO"
```

The HTAB *Statement*

```
HTAB 10
HTAB N
HTAB 41 - LEN (S$)
```

HTAB **moves cursor to a specified horizontal position**

HTAB (for "horizontal tab") moves the cursor to a specified horizontal position from the beginning of the current output line. The expression following the keyword HTAB specifies the position to which the cursor is to be moved. If this value is a real number, Applesoft truncates it to the next lowest integer.

TAB **function:** see above

HTAB can move cursor in either direction

Unlike the TAB function, which moves the cursor in a forward direction only, the HTAB statement can move the cursor in either direction to a specified horizontal position. For example, the program

```
 5 HOME                          —clear text from screen
10 HTAB 6 : PRINT "IS THE ";
                                 —display IS THE starting at
                                   column 6
15 FOR Z = 1 TO 500: NEXT Z
                                 —delay loop so user can see the
                                   order and position of text
                                   display
20 HTAB 1 : PRINT "THIS ";
                                 —display THIS at column 1
25 FOR Z = 1 TO 500 : NEXT Z
                                 —another delay loop
30 HTAB 13 : PRINT "PROPER ORDER"
                                 —display PROPER ORDER at
                                   column 13
```

will display the following on the screen:

```
THIS IS THE PROPER ORDER
```

If you want to use HTAB to display several text items on the same line, you need a semicolon at the end of each PRINT statement, as in the program above, to avoid starting a new line.

text window: see below

If there is a text window set, the specified tab position is interpreted relative to the left edge of the window. However, HTAB behaves as though there were 40 columns in each line of the window, regardless of the actual width to which the window has been set; that is, position 1 is considered to be the leftmost column of the current line, position 41 the leftmost column of the next line, position 81 the leftmost column of the line after that, and so on. If the cursor is advanced past the right edge of the screen or text window, it returns to the beginning of the next line and continues advancing.

HTAB can carry the cursor outside the boundaries of the text window, but only long enough to display one character; the cursor then returns to the left edge of the window.

80-Column Text Card: see *Apple IIe Owner's Manual, Apple IIe 80-Column Text Card Manual*

80-Column Text Card Users: HTAB is designed to operate with a 40-column screen only. If you attempt to advance the cursor beyond column 40, it will "wrap around" to the next line, even if you have the Apple IIe 80-Column Text Card installed and running in "active 80" mode. To tab to a position between columns 41 and 80, use

POKE **statement:** see Section 7.1.2

```
POKE 36, XX
```

where XX the number of the column to which you want to tab. See Section F.1 and the *80-Column Text Card Manual* for more information.

The column number specified to HTAB must be in the range 0 to 255 or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

A value of 0 moves the cursor to 256 positions from the beginning of the current line.

Many programmers find HTAB to be more convenient to use than TAB, because it is an independent statement and need not be embedded in a PRINT statement. This makes it easier to change, if necessary, during program development.

The VTAB *Statement*

```
VTAB 10
VTAB N
VTAB 25 - H%
```

VTAB **moves cursor to a specified vertical position**

VTAB (for "vertical tab") moves the cursor vertically to a specified line on the screen. The expression following the keyword VTAB specifies the line to which the cursor is to be moved. If the value of this expression is a real number, Applesoft truncates it to the next lowest integer.

The top line of the screen is line 1; the bottom line is line 24. VTAB may move the cursor either up or down on the screen, but never to the left or right; it remains at the same horizontal position as before the move. For example,

```
10 HOME                    —clear text from screen
20 VTAB 6                  —move cursor to line 6
30 PRINT "LINE 6"          —display imaginative message
```

```
40 FOR Z = 1 TO 500 : NEXT Z
```
—delay loop so user can see the order and position of text display

```
50 VTAB 18
```
—move cursor to line 18

```
60 PRINT "LINE 18"
```
—display another imaginative message

```
70 FOR Z = 1 TO 500 : NEXT Z
```
—another delay loop

```
80 VTAB 12
```
—move cursor to line 12

```
90 PRINT "THE MIDDLE"
```
—display last message

Text window ignored

VTAB ignores the setting of the text window, if any. The specified line number is always taken to refer to the entire screen.

The line number specified to VTAB must be in the range 1 to 24 or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

If VTAB moves the cursor to a line below the bottom of the text window, all subsequent text will be displayed on that same line.

The POS *Function*

POS **yields current horizontal cursor position**

The POS (for "position") function yields the current horizontal position of the cursor, relative to the left edge of the screen or text window. The value yielded is in the range 0 to 39 (0 to 79 if the Apple IIe 80-Column Text Card is installed and running in "active 80" mode). A value of 0 represents the left edge of the screen or window.

Argument required but ignored

Strange but True: The argument given to POS is ignored, and has no effect on the operation of the function. However, you can't leave it out—you must include an argument expression of some kind to "keep the parentheses apart." What you use for an argument expression doesn't matter, but if Applesoft can't evaluate it as a legal expression, you'll get an error halt.

POS, TAB, and HTAB disagree

A Difference of Opinion: Notice that POS numbers columns beginning with 0, whereas TAB and HTAB number them beginning with 1. Thus, assuming the cursor is at the beginning of a line, the statement

```
PRINT TAB(10); POS (0)   —tab to column 10 and display
                           position
```

will display the value 9, and

```
HTAB 43 : PRINT POS (X)
                         —tab to column 43 and display
                           position
```

will display 2 (since HTAB 43 tabs to the third column of the next display line). Notice in the second case that the value of variable X makes no difference.

The INVERSE *Statement*

```
INVERSE
```

INVERSE **displays text in black-on-white**

NORMAL **statement:** see below

The INVERSE statement causes subsequent text output to be displayed in black-on-white instead of the usual white-on-black (where "white" means the phosphor color of your display, whatever that is). The normal white-on-black display can be restored with the NORMAL statement. For example,

```
10 INVERSE                  —set inverse display
20 PRINT "BLACK-ON-WHITE"
                            —display BLACK-ON-
                              WHITE in black-on-white
30 NORMAL                   —restore normal display
40 PRINT "WHITE-ON-BLACK"
                            —display WHITE-ON-
                              BLACK in white-on-black
```

PRINT **statement:** see Section 5.2.2

INVERSE affects only subsequent output characters sent to the screen with PRINT statements. It has no effect on characters already on the screen or on keyboard input "echoed" to the screen.

Don't Overdo It: INVERSE is most effective when you use it sparingly.

The FLASH *Statement*

FLASH

FLASH causes text to flash on the screen

The FLASH statement causes subsequent text output to alternate approximately twice a second between black-on-white and the usual white-on-black (where "white" means the phosphor color of your display, whatever that is). The normal white-on-black display can be restored with the NORMAL statement. For example,

NORMAL statement: see below

```
10 FLASH                    —set flashing display
20 PRINT "FLASHY"           —display flashy FLASHY
30 NORMAL                   —restore normal display
40 PRINT "DRAB"             —display drab DRAB
```

PRINT statement: see Section 5.2.2

FLASH affects only subsequent output characters sent to the screen with PRINT statements. It has no effect on characters already on the screen or on keyboard input "echoed" to the screen.

ASCII codes: see Section 4.2.1 and Appendix C

FLASH doesn't work on characters with ASCII codes above 95, the most important of which are the lowercase letters; instead of making them flash, it turns them into gibberish. FLASH doesn't work at all if you have the Apple IIe 80-Column Text Card installed and running in "active 80" mode.

80-Column Text Card: see *Apple IIe Owner's Manual, 80-Column Text Card Manual*

A Little Dab'll Do Ya: FLASH is most effective when you use it very sparingly. Reserve it for only the most important messages or unusual uses. Cavalier use of FLASH has been known to drive users to delirium.

The NORMAL *Statement*

```
NORMAL
```

NORMAL displays text in white-on-black

The NORMAL statement causes subsequent text output to be displayed in the usual white-on-black (where "white" means the phosphor color of your display, whatever that is). It is usually used to cancel the effects of the INVERSE or FLASH statement. For example,

INVERSE and FLASH statements: see above

```
10 INVERSE                    —set inverse display
20 PRINT "BLACK-ON-WHITE"
                              —display BLACK-ON-
                                WHITE in black-on-white
30 NORMAL                     —restore normal display
40 PRINT "WHITE-ON-BLACK"
                              —display WHITE-ON-
                                BLACK in white-on-black
50 FLASH                      —set flashing display
60 PRINT "FLASHY"             —display flashy FLASHY
70 NORMAL                     —restore normal display
80 PRINT "DRAB"               —display drab DRAB
```

PRINT statement: see Section 5.2.2

NORMAL affects only subsequent output characters sent to the screen with PRINT statements. It has no effect on characters already on the screen.

The SPEED= *Statement*

```
SPEED= 255
SPEED= X
SPEED= Z - 6 * F
```

SPEED= sets rate of text output

The SPEED= statement sets the rate at which output characters are sent to the display screen or other output device (such as a printer). The slowest rate is 0; the fastest is 255. The normal speed setting (if you don't do anything to change it) is 255. For example,

```
10 SPEED= 0                   —set slowest possible speed
20 PRINT "THE TORTOISE"
                              —display THE TORTOISE
                                slowly
30 SPEED= 255                 —restore normal speed
40 PRINT "THE HARE"           —display THE HARE quickly
```

SPEED isn't a variable

Notice that the equal sign is part of the keyword SPEED= ; it doesn't represent an assignment to a variable named SPEED. A statement such as

```
LET SPEED = X
```

will cause a syntax error. The only way to find out the current speed setting is to keep track of it yourself with a variable, as in the following example:

```
10 LET X = 250              —set initial value for speed
20 SPEED= X                 —set speed to value of X
30 PRINT "CURRENT SPEED IS "; X
                            —display current speed
40 LET X = X - 25           —decrease value of X by 25
50 IF X >= 0 THEN GOTO 20
                            —repeat until X becomes
                              negative
60 SPEED= 255 : END         —X is too low; end the program
```

The speed setting is not reset to its normal value by RUN, CLEAR or CONTROL-RESET.

The speed setting specified to SPEED= must be in the range 0 to 255 or the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

The Text Window

The "window" within which text is displayed and scrolled on the screen can be set to less than the full screen through the magic of the POKE statement. See Section F.1 for details and the *Apple IIe Reference Manual* for a more technical discussion.

POKE **statement:** see Section 7.1.2

5.2.5 *Miscellaneous Output Facilities*

This section covers Applesoft's facilities for dealing with the remaining output features of the Apple IIe: the built-in speaker, annunciator outputs, utility strobe, and cassette tape output. Most of these features are controlled by means of PEEK and POKE; details can be found in Appendix F, "Peeks, Pokes, and Calls." The annunciators and utility strobe are seldom used, and are mentioned here just for the sake of completeness.

PEEK **function:** see Section 7.1.1

POKE **statement:** see Section 7.1.2

Controlling The Speaker

CONTROL-G sounds the "bell"

The Apple IIe has a small, built-in speaker that you can use to add sound to your programs. The easiest way to use it is by sending the ASCII "bell" character (CONTROL-G, ASCII code 7) to the display screen. This causes the computer to emit a short "beep."

Historical Note: ASCII code 7 was originally used to ring a small bell on teletype machines, to let the teletype operator know that a message was coming in. On the Apple IIe it sends a 1-kilohertz tone (1000 cycles per second) to the computer's speaker for 1/10 second.

Here's a program to ring the computer's "bell" a number of times specified by the user:

```
10 PRINT "ENTER A NUMBER FROM 1 TO 9
     (S TO STOP):";   —prompt user for input
20 GET A$             —accept single character from
                        keyboard
30 IF A$ = "S" THEN END
                      —stop if user typed S
40 IF VAL (A$) < 1 THEN 20
                      —if character typed is out of
                        range then try again
50 FOR X = 1 TO VAL (A$)
                      —loop requested number of
                        times
60 PRINT CHR$ (7)     —sound "bell"
70 NEXT X             —loop back to 50
80 PRINT              —leave a blank line
90 GOTO 10            —start again
```

The only other way to produce sound from the speaker is with a PEEK or POKE to address -16336. This causes the speaker to emit a single "click." By combining such clicks in the appropriate patterns and frequencies, you can produce musical tones and a variety of other sounds. Experiment for yourself!

For technical information on the built-in speaker, see the *Apple IIe Reference Manual.*

Annunciator Output

annunciators: see *Apple IIe Reference Manual*

The Apple IIe has four *annunciator* outputs, which are pins of the hand control connector on which electrical impulses can be transmitted. The signals on these pins are most commonly used to control devices such as lamps and relays connected to the computer through the hand control connector. The annunciator outputs can be turned on and off with PEEK or POKE to the appropriate addresses; see Section F.4 for details and the *Apple IIe Reference Manual* for further technical information.

The Utility Strobe

utility strobe: see *Apple IIe Reference Manual*

The Apple IIe's *utility strobe* is a pin of the hand control connector that can be triggered to send an electrical impulse lasting one-half microsecond. Like the annunciators, it can be used to control a variety of devices connected to the computer through the hand control connector. The utility strobe can be triggered with a PEEK or POKE to address -16320; see Section F.4 for details and the *Apple IIe Reference Manual* for further technical information.

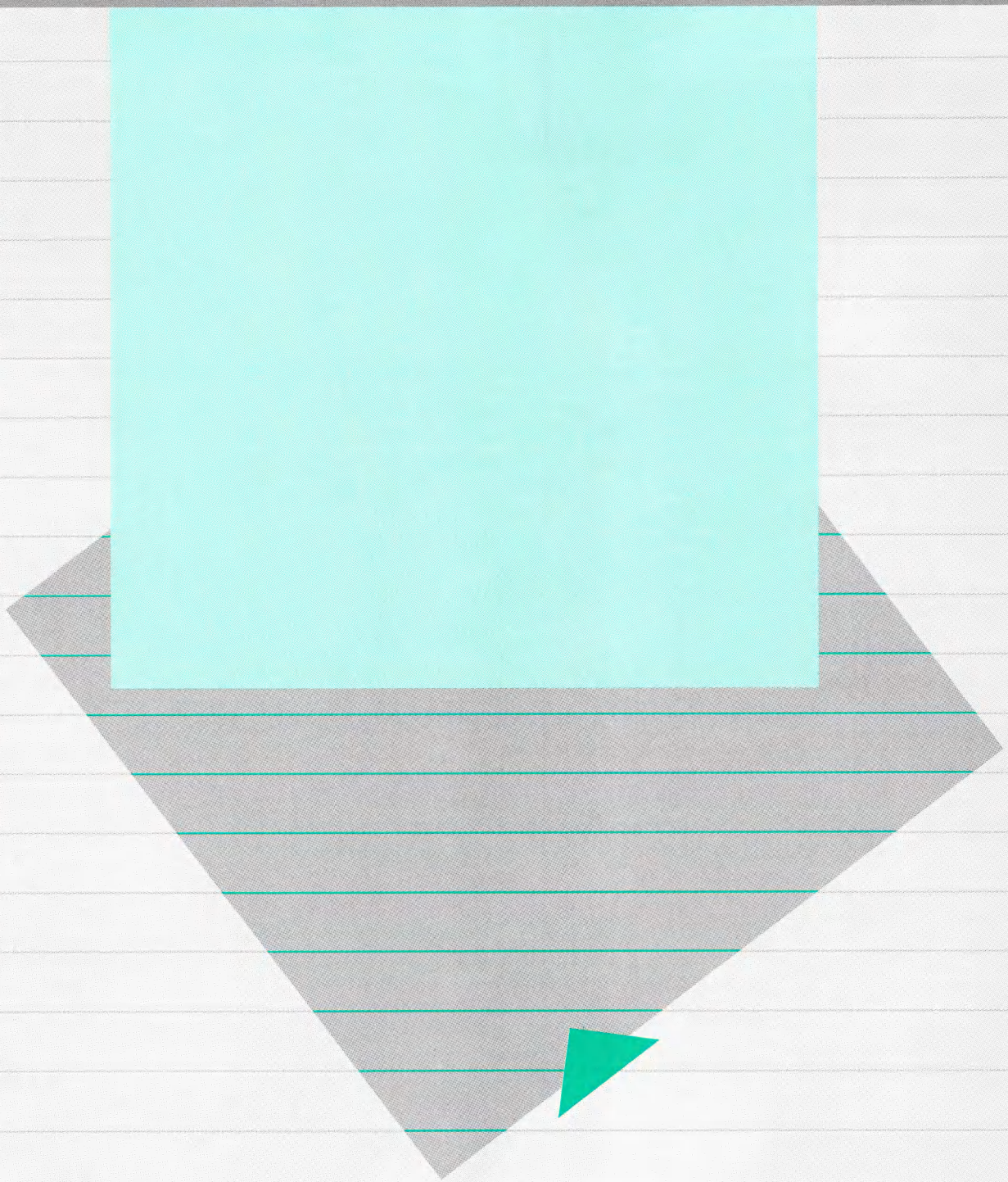
Cassette Output

SAVE **command:** see Section 1.2.5 and Appendix M

STORE **statement:** see Appendix M

Two Applesoft statements, SAVE and STORE, can be used to write information to a cassette tape recorder. SAVE writes the Applesoft program currently in memory to tape; STORE writes the contents of an integer or real array. For details, see Appendix M, "If You Have a Cassette Recorder."

Graphics



Graphics

This chapter describes Applesoft's facilities for creating, changing, displaying, and storing both low- and high-resolution graphic designs.

low-resolution graphics: see Section 6.1

Section 6.1, "Low-Resolution Graphics," deals with 16-color graphics on a 40-by-48 grid.

high-resolution graphics: see Section 6.2

Section 6.2, "High-Resolution Graphics," deals with 6-color graphics on a 280-by-192 grid.

shape tables: see Section 6.3

Section 6.3, "Shape Tables," discusses the use of *shape tables* for animation sequences.

6.1 Low-Resolution Graphics

The low-resolution graphics screen consists of 1920 blocks (40 columns by 48 rows) in 16 colors. This section describes the facilities available in Applesoft for using low-resolution graphics:

GR **statement:** see Section 6.1.1

- The GR statement instructs Applesoft to begin displaying low-resolution graphics.

COLOR= **statement:** see Section 6.1.2

- The COLOR= statement controls the colors displayed on the screen.

PLOT **statement:** see Section 6.1.3

- The PLOT statement plots individual blocks on the screen.

HLIN **statement:** see Section 6.1.4

- The HLIN statement draws horizontal lines.

VLIN **statement:** see Section 6.1.5

- The VLIN statement draws vertical lines.

SCRN **function:** see Section 6.1.6

- The SCRN function determines what color is currently displayed at any position of the screen.

6.1.1 *The GR Statement*

```
GR
```

GR displays low-resolution graphics

The GR (for "graphics") statement instructs the computer to display low-resolution graphics. If the screen has been displaying text, it is changed from 40 (or 80) columns by 24 lines of text to 40 columns by 40 rows of graphics, with space for four lines of text at the bottom. (Full text display can be restored with the TEXT statement.) GR clears the screen to black, moves the text cursor to the beginning of the bottom line (line 24), clears any text window that may have been set, and sets the low-resolution display color to 0 (black).

TEXT statement: see Section 5.2.4

text window: see Section 5.2.4

After executing a GR statement, you can convert the display to full-screen graphics (a 40-by-48 grid with no space for text) with the statement

POKE statement: see Section 7.1.2

```
POKE -16302,0
```

This statement will change the bottom four lines of text to eight rows of colored blocks. To clear these rows to black, add

```
CALL -1998
```

Notice that the POKE statement above must be executed after GR. If you execute the POKE first, GR will reset the screen to mixed graphics and text.

high-resolution page 2: see Section 6.2.2

If you execute a GR statement while displaying high-resolution page 2, GR clears its usual screenful of memory but leaves you looking at page 2 of low-resolution graphics and text. To avoid this problem, always use the TEXT statement before switching from high-resolution page 2 to low resolution.

For more information...

See Section F.3 for more information on the use of the various text and graphics memory pages. See the *Apple IIe Reference Manual* for further technical information on the Apple IIe's graphics display capabilities.

6.1.2 *The* COLOR= *Statement*

```
COLOR= 12
COLOR= C(J)
COLOR= (X - 4) / 16
```

COLOR= sets low-resolution display color

The COLOR= statement sets the display color for plotting low-resolution graphics. There are 16 colors available, represented by numbers from 0 to 15 as shown in Table 6-1. When you enter low-resolution graphics, the GR statement sets the display color to black (0).

Table 6-1 Color Codes for Low-Resolution Graphics

Code	Color	Code	Color
0	black	8	brown
1	magenta	9	orange
2	dark blue	10	grey-2
3	violet	11	pink
4	dark green	12	green
5	grey-1	13	yellow
6	medium blue	14	aqua
7	light blue	15	white

If you're using a monochrome display (black-and-white, or some other single phosphor color), the different colors will appear on your screen as various patterns of shading.

The following short program displays each of the 16 available colors in a horizontal bar across the screen:

```
10 GR                          —display low-resolution
                                graphics
20 FOR X = 0 TO 15             —execute loop for each color
30 COLOR= X                    —set next color
40 HLIN 0, 39 AT X * 2
                               —draw a bar of this color across
                                the screen, leaving a blank
                                row above it
50 NEXT X                      —go back for next color
```

HLIN statement: see Section 6.1.4

Notice that the equal sign is part of the keyword COLOR=; it doesn't represent an assignment to a variable named COLOR. A statement such as

COLOR isn't a variable

```
LET COLOR= X
```

will cause a syntax error. The only way to find out the current display color is to keep track of it yourself with a separate variable, as in the example above.

You can specify a color code higher than 15, but the series of color values simply repeats. That is, 16 is equivalent to 0, 18 is equivalent to 2, 35 is equivalent to 3, and so on. However, a color value less than 0 or greater than 255 will stop the program with the message

```
?ILLEGAL QUANTITY ERROR
```

6.1.3 *The* PLOT *Statement*

```
PLOT 20, 12
PLOT X - 6, Y + 2
PLOT THETA * 40 / (2*PI), 24 -
   (SIN(THETA) * 23)
```

PLOT **draws a single block**

low-resolution display color: see Section 6.1.2

The PLOT statement places a block of the current low-resolution display color at a specified position on the screen. The first expression following the keyword PLOT specifies the column in which the block is to be plotted (numbered 0 to 39, from left to right); the second expression, separated from the first by a comma, designates the row (numbered 0 to 39 for mixed text and graphics, 0 to 47 for full-screen graphics, from top to bottom). For example, the following program plots a block of pink in column 20, row 2 of the screen:

```
10 GR                    —display low-resolution
                           graphics
20 COLOR= 11             —set display color to pink
30 PLOT 20, 2            —plot a block of pink in column
                           20, row 2
```

Figure 6-1 shows the system of coordinates used to designate positions on the low-resolution graphics screen. Position 0,0 is the top-left corner and position 39,0 the top-right. When displaying mixed graphics and text, the bottom-left corner is position 0,39 and the bottom-right is 39,39; in full-screen graphics, the bottom-left corner is 0,47 and the bottom-right is 39,47.

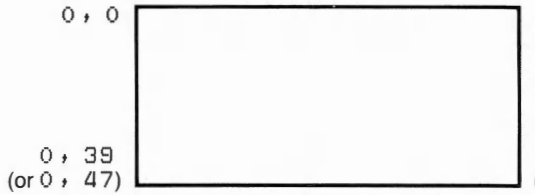


Figure 6-1 Screen Coordinates for Low-Resolution Graphics

If Applesoft is displaying mixed graphics and text and the plotting coordinates designate a row from 40 to 47, a text character will be displayed at the specified coordinates instead of a block of color. The particular character displayed depends on the current low-resolution display color. Here's a program to demonstrate this effect:

```
10 GR                          —display mixed low-resolution
                                graphics
20 FOR Y = 0 TO 47             —loop over all screen rows
30 FOR X = 0 TO 39             —loop over all screen columns
40 COLOR = X                   —use color corresponding to col-
                                umn number (colors 0 to 15 will
                                repeat after column 15)
50 PLOT X, Y                   —plot a block at column X, row Y
60 NEXT X                      —loop to next column
70 NEXT Y                      —loop to next row
```

Try changing line 10 to

```
10 TEXT
```

to see the effect on the full screen.

A column coordinate outside the range 0 to 39 or a row coordinate outside the range 0 to 47 will cause the program to halt with the message

```
?ILLEGAL QUANTITY ERROR
```

6.1.4 *The* HLIN *Statement*

```
HLIN 5, 20 AT 35
HLIN X, Y AT Z
HLIN Q - 3, J * 58 AT V%
```

HLIN **draws a horizontal line**

low-resolution display color: see Section 6.1.2

The HLIN (for "horizontal line") statement draws a horizontal line on the screen in the current low-resolution display color. The two expressions following the keyword HLIN, separated by a comma, designate the columns in which the line is to begin and end; the expression following the keyword AT specifies the row in which the line is to be

drawn. The first end point may be less than, equal to, or greater than the second. For example,

```
10 GR                          —display low-resolution
                                graphics
20 COLOR= 4                    —set color to dark green
30 HLIN 10, 30 AT 20           —draw a horizontal green line in
                                row 20 from column 10 to col-
                                umn 30
```

If you use HLIN while displaying text instead of graphics, or with a row coordinate from 40 to 47 while displaying mixed graphics and text, Applesoft will display a row of characters instead of a bar of color. For example, if line 10 above were changed to

```
10 TEXT                        —display text instead of graphics
```

the result would be a row of dollar signs instead of a bar of dark green. In most cases, when you see patterns like these on your screen it means you forgot to include a GR statement.

A column coordinate outside the range 0 to 39 or a row coordinate outside the range 0 to 47 will cause the program to halt with the message

```
?ILLEGAL QUANTITY ERROR
```

The HLIN statement has no visible effect if you use it while displaying high-resolution graphics.

6.1.5 *The* VLIN *Statement*

```
VLIN 5, 20 AT 35
VLIN X, Y AT Z
VLIN Q - 3, J * 58 AT V%
```

VLIN **draws a vertical line**

low-resolution display color: see Section 6.1.2

The VLIN (for "vertical line") statement draws a vertical line on the screen in the current low-resolution display color. The two expressions following the keyword VLIN, separated by a comma, designate the rows in which the line is to begin and end; the expression

following the keyword AT specifies the column in which the line is to be drawn. The first end point may be less than, equal to, or greater than the second. For example,

```
10 GR                          —display low-resolution
                                graphics
20 COLOR= 4                    —set color to dark green
30 VLIN 10, 30 AT 20           —draw a vertical green line in
                                column 20 from row 10 to row
                                30
```

If you use VLIN while displaying text instead of graphics, or if part of the line being drawn goes beyond row 39 while displaying mixed graphics and text, Applesoft will display text characters instead of blocks of color. For example, if line 10 above were changed to

```
10 TEXT                        —display text instead of graphics
```

the result would be a row of flashing D's and a dollar sign instead of a bar of dark green.

A column coordinate outside the range 0 to 39 or a row coordinate outside the range 0 to 47 will cause the program to halt with the message

```
?ILLEGAL QUANTITY ERROR
```

The VLIN statement has no visible effect if you use it while displaying high-resolution graphics.

6.1.6 *The SCRN Function*

SCRN reads the color at a designated screen position

The SCRN (for "screen") function reads the color currently displayed at a designated position on the low-resolution graphics screen. This function takes two arguments, the first specifying the column and the second the row of the desired position. It yields a number from 0 to 15 representing the color displayed at that position. For example, the expression

color codes: see Table 6-1, Section 6.1.2

```
SCRN (5, 9)
```

yields the code for the color displayed at column 5, row 9.

The SCRN function is not intended for use with high-resolution graphics.

For Experts Only—Strange Extensions: Although the ordinary limits for coordinates on the low-resolution graphics screen are 39 and 47, SCRN will actually accept values up to 47 for both arguments. But if the column parameter is greater than the usual limit of 39, odd things happen. The code yielded by SCRN gives the color for the block whose column is the designated column minus 40, and whose row is the designated row plus 16.

If the row-plus-16 number is in the range 40 through 47, and if mixed graphics and text are being displayed, then the code yielded is not a color code, but is related to the text character at that position in the text area below the graphics (see "For Experts Only—Reading the Text Screen," below).

If the row-plus-16 number is in the range 48 to 63, SCRN yields a result whose meaning is beyond the ken of mere mortals.

For Experts Only—Reading the Text Screen: When text is being displayed, SCRN yields numbers in the range 0 to 15 whose value is either the high-order four bits (if the row number is odd) or the low-order four bits (if the row number is even) of the character in column C + 1 and row (R + 1) / 2, where C and R are the column and row numbers given as arguments to SCRN. Thus the following expression will yield the character at position X,Y:

```
CHR$ ( SCRN (X - 1, 2 * (Y - 1) + 1) * 16
   + SCRN (X - 1, 2 * (Y - 1)) )
```

High-Resolution Graphics

6.2

There are two separate regions in the Apple IIe's memory, designated page 1 and page 2, that can be used for displaying high-resolution graphics. Each consists of 53,760 points (280 columns by 192 rows), which can be displayed on the screen in 6 colors. This section describes the facilities available in Applesoft for using high-resolution graphics:

HGR **statement:** see Section 6.2.1

- The HGR statement instructs Applesoft to begin displaying page 1 of high-resolution graphics.

HGR2 **statement:** see Section 6.2.2

- The HGR2 statement instructs Applesoft to begin displaying page 2 of high-resolution graphics.

HCOLOR= **statement:** see Section 6.2.3

- The HCOLOR= statement controls the colors displayed on the high-resolution screen.

HPLOT **statement:** see Section 6.2.4

- The HPLOT statement plots individual points and lines on the high-resolution screen.

protecting programs and graphics: see Section 6.2.5

Section 6.2.5 tells how to protect your programs and high-resolution graphics from overwriting each other in the computer's memory.

6.2.1 *The* HGR *Statement*

```
HGR
```

HGR displays high-resolution graphics page 1

The HGR (for "high-resolution graphics") statement instructs Applesoft to display page 1 of high-resolution graphics. If the screen has been displaying text, it is changed from 40 (or 80) columns by 24 lines of text to 280 columns by 160 rows of high-resolution graphics, with space for four lines of text at the bottom. The graphics area of the screen is cleared to black; the high-resolution display color is not affected. HGR doesn't affect the contents of the text screen, the setting of the text window, or the location of the text cursor; the cursor will not be visible unless it is in one of the bottom four lines of the screen.

high-resolution display color: see Section 6.2.3

After executing an HGR statement, you can convert the display to full-screen graphics (a 280-by-192 grid with no space for text) with the statement

POKE **statement:** see Section 7.1.2

```
POKE -16302, 0
```

This statement will change the bottom four lines of text to high-resolution graphics. To return to mixed graphics and text, use

```
POKE -16301, 0
```

Notice that the first POKE statement above must be executed after HGR. If you execute the POKE first, HGR will reset the screen to mixed graphics and text.

TEXT **statement:** see Section 5.2.4

The TEXT statement will return to text display with the text window set to the full screen and the cursor at the bottom of the screen. To turn off high-resolution graphics and return to text display with the text window and cursor intact, use the statement

```
POKE -16303, 0
```

protecting programs and graphics: see Section 6.2.5

If you intend to use HGR with an Applesoft program longer than about 6000 (decimal) bytes, see Section 6.2.5 on how to protect your program and graphics from overwriting each other.

Warning

If you use the reserved word HGR as the first three characters of a variable name, the HGR statement may be executed before a syntax error is detected. For example, executing the statement

```
HGRIP = 4
```

will unexpectedly turn on high-resolution graphics and may destroy part of your program.

For more information...

See Section F.3 for more information on the use of the various text and graphics memory pages, and Section H.1 for the memory locations occupied by the high-resolution graphics pages. See the *Apple IIe Reference Manual* for further technical information on the Apple IIe's graphics display capabilities.

6.2.2 *The* HGR2 *Statement*

```
HGR2
```

HGR2 displays high-resolution graphics page 2

high-resolution display color: see Section 6.2.3

text window: see Section 5.2.4

The HGR2 (for "high-resolution graphics, page 2") statement instructs Applesoft to display page 2 of high-resolution graphics. If the screen has been displaying text, it is changed from 40 (or 80) columns by 24 lines of text to 280 columns by 192 rows of high-resolution graphics. The screen is cleared to black; the high-resolution display color is not affected. HGR2 doesn't affect the contents of the text screen, the setting of the text window, or the location of the text cursor.

TEXT statement: see Section 5.2.4

The TEXT statement will return to text display with the text window set to the full screen and the cursor at the bottom of the screen. To turn off high-resolution graphics and return to text display with the text window and cursor intact, use the statements

POKE statement: see Section 7.1.2

```
POKE -16300, 0        —switch from page 2 to page 1
POKE -16303, 0        —switch from graphics to text
```

After executing an HGR2 statement, you can convert the display to mixed graphics and text (a 280-by-160 grid with four lines of text at the bottom) with the statement

```
POKE -16301, 0
```

However, when you use this statement while displaying high-resolution page 2, the four lines of text will be taken from text page 2 instead of the usual page 1. Since Applesoft uses the same memory locations allocated to text page 2 for program storage, you'll end up displaying garbage in the bottom four lines of your screen. For this reason, most programmers avoid mixing graphics and text when using high-resolution page 2.

protecting your program: see Section 6.2.5

If you intend to use HGR2 with an Applesoft program longer than about 14000 (decimal) bytes, see Section 6.2.5 on how to protect your program and graphics from overwriting each other.

Warning

If you use the reserved word HGR2 as the first four characters of a variable name, the HGR2 statement may be executed before a syntax error is detected. For example, executing the statement

```
HGR2PIECES = 4
```

will unexpectedly turn on high-resolution graphics and may destroy part of your program.

For more information...

See Section F.3 for more information on the use of the various text and graphics memory pages, and Section H.1 for the memory locations occupied by the high-resolution graphics pages. See the *Apple IIe Reference Manual* for further technical information on the Apple IIe's graphics display capabilities.

6.2.3 *The* HCOLOR= *Statement*

```
HCOLOR= 6
HCOLOR= C(J)
HCOLOR= (X - 4) / 8
```

HCOLOR= **sets high-resolution display color**

The HCOLOR= (for "high-resolution color") statement sets the display color for plotting high-resolution graphics. There are 6 colors available, represented by numbers from 0 to 7, as shown in Table 6-2.

Table 6-2 Color Codes for High-Resolution Graphics

Code	Color
0	black-1
1	green
2	violet
3	white-1
4	black-2
5	orange
6	blue
7	white-2

If you're using a monochrome display (black-and-white, or some other single phosphor color), the different colors will appear on your screen as various patterns of shading.

The high-resolution display color is not affected by HGR, HGR2, or RUN. Until your program executes an HCOLOR= statement, the display color for high-resolution graphics is indeterminate.

Notice that the equal sign is part of the keyword HCOLOR=; it doesn't represent an assignment to a variable named HCOLOR. A statement such as

```
LET HCOLOR= X
```

will cause a syntax error. The only way to find out the current display color is to keep track of it yourself with a separate variable.

Curious Behavior: As you wander deeper into the recesses of the Apple IIe's graphics system, you'll begin to notice that the colors in high-resolution graphics don't always act as you might expect. For example, carefully drawn vertical lines may refuse to be visible, a white line crossing a field of green may leave jagged blocks of orange in its wake, or a point plotted with HCOLOR= 3 (white-1) may look blue if its column coordinate is even, green if the column coordinate is odd, and white only if a point is plotted in the next column as well. These strange phenomena are a result of the way the Apple IIe's high-resolution graphics features interact with the color circuitry in your television set. See the *Apple IIe Reference Manual* for further explanation.

If you specify a color code higher than 7, your program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

6.2.4 *The* HPLOT *Statement*

```
HPLOT 140, 80
HPLOT X - 16, Y + 12 TO X + 16, Y -
   12
HPLOT 70,40 TO 210,40 TO 210,120 TO
   70,120 TO 70,40
HPLOT TO THETA * 280 / (2*PI), 96 -
   (SIN(THETA) * 95)
```

HPLOT plots high-resolution points and lines

high-resolution display color: see Section 6.2.3

The HPLOT (for "high-resolution plot") statement plots points and lines on the high-resolution graphics screen in the current high-resolution display color. The first expression in each pair specifies a column (numbered 0 to 279, from left to right); the second expression, separated from the first by a comma, designates a row (numbered 0 to 159 for mixed text and graphics, 0 to 191 for full-screen graphics, from top to bottom). For example, the following program plots a white point at column 100, row 50 of the screen:

```
10 HGR                    —display high-resolution
                            graphics
20 HCOLOR= 3              —set display color to white-1
30 HPLOT 100, 50          —plot a point at column 100, row
                            50
```

Figure 6-2 shows the system of coordinates used to designate positions on the high-resolution graphics screen. Position 0,0 is the top-left corner and position 279,0 the top-right. When displaying mixed graphics and text, the bottom-left corner is position 0,159 and the bottom-right is 279,159; in full-screen graphics, the bottom-left

corner is 0,191 and the bottom-right is 279,191.

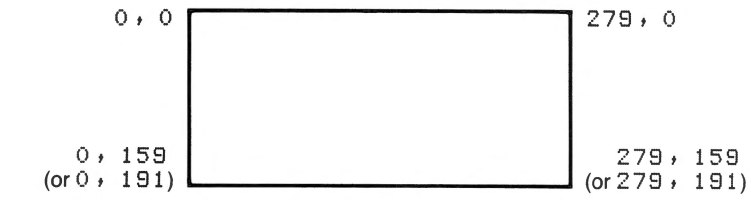


Figure 6-2 Screen Coordinates for High-Resolution Graphics

To draw a line with HPLOT, specify the starting and ending points, separated by the keyword TO. The next example draws a white line across the screen:

```
10 HGR                          —display high-resolution
                                  graphics
20 HCOLOR= 3                    —set display color to white-1
30 HPLOT 0, 50 TO 279, 50
                                —draw a line across row 50
```

You can draw a series of connected lines in the same HPLOT statement by using a series of TO clauses. Each line will begin where the last one ended. The following program, for example, draws a rectangle, as illustrated in Figure 6-3:

```
10 HGR                          —display high-resolution
                                  graphics
20 HCOLOR= 3                    —set display color to white-1
30 HPLOT 70,40 TO 210,40 TO 210,120
     TO 70,120 TO 70,40
                                —draw a rectangle
```

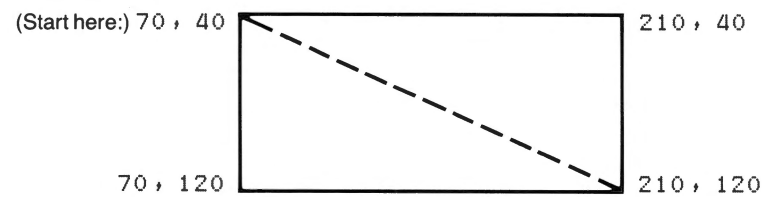


Figure 6-3 Drawing a Rectangle with HPLOT

You can extend the series of lines almost indefinitely within the same HPLOT statement, subject only to the limit of 239 characters in a program line.

You can also continue from wherever the last HPLOT statement ended, by writing the keyword TO immediately after the word HPLOT. For example, adding the line

```
40 HPLOT TO 210, 120   —continue drawing from last point
```

to the previous program will cause it to draw in the diagonal of the rectangle, represented by the dashed line in Figure 6-3. Applesoft assumes that the starting point (which ordinarily would have appeared between the words HPLOT and TO) is the last point plotted.

HCOLOR= **statement:** see Section 6.2.3

The color of the new line drawn by HPLOT TO is the same as that of the last point plotted. Even if you insert a new HCOLOR= statement between lines 30 and 40, the line drawn by the HPLOT TO in program line 40 will appear in the same color as those drawn in line 30.

To change the color of the line, use a whole new HPLOT:

```
35 HCOLOR= 6              —change color to blue
40 HPLOT 70, 40 TO 210, 120
                          — continue drawing from last point
```

If the screen is displaying mixed text and graphics, an attempt to plot a point whose row coordinate is in the range 160 to 191 will have no visible effect. However, if you draw a line either beginning or ending in rows 160 to 191, Applesoft will display as much of the line as it can. If you later switch to full-screen graphics with POKE -16302,0 the hidden portion of the line will appear.

Warning

Be sure to precede HPLOT by either HGR or HGR2 or you will write over lots of memory, including your program and variables.

If the column coordinate given to HPLOT is outside the range 0 to 279, or the row coordinate outside the range 0 to 191, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

6.2.5 *Protecting High-Resolution Graphics*

Apple IIe memory allocation: see Section H.1

The two high-resolution graphics pages lie pretty much in the center of things: page 1 resides at memory addresses 8192 to 16383 (hexadecimal $2000 to $3FFF) and page 2 at addresses 16384 to 24575 (hexadecimal $4000 to $5FFF). Because Applesoft

HGR **statement:** see Section 6.2.1

program storage begins at location 2048 (hexadecimal $800), it's easy for your program and graphics to get in each other's way. For example, if you're using the HGR statement to display page 1 of high-resolution graphics, you have only 6144 bytes of program and variable space (8192 minus 2048) before your program overwrites the graphics area. This section tells how to prevent them from colliding, causing untold mayhem and destruction.

HIMEM: **statement:** see Section 7.2.1

One way to protect your program and graphics from each other is to use the HIMEM: statement to set the upper limit of program memory at 8192. This is a reasonable method to use for short programs; but Applesoft tends to use a lot of memory, and longer programs would soon run out of space.

HGR2 **statement:** see Section 6.2.2

Another method that allows the program a bit more breathing room is to use the second page of graphics instead of the first (HGR2 instead of HGR). This has the benefit of starting the graphics at a higher memory location, so you can set HIMEM: to 16384 instead of 8192, allowing 14336 bytes (16384 minus 2048) for your program and variable space. The disadvantage of this method is that you lose the four lines of text at the bottom of the screen, which are available with HGR but not with HGR2.

POKE **statement:** see Section 7.1.2

A third method, probably the best for long programs with lots of variables, is to use the wizardry of the POKE statement to change the start, instead of the end, of Applesoft's program storage space. The following statements will start program and variable storage above graphics page 1, beginning at address 16384 (hexadecimal $4000):

```
POKE 103, 1
POKE 104, 64
POKE 16384, 0
```

These statements will start program and variable storage above high-resolution page 2, beginning at address 24576 (hexadecimal $6000):

```
POKE 103, 1
POKE 104, 96
POKE 24576, 0
```

No matter where you start program space in memory, your next command should be

NEW **command:** see Section 1.2.1

```
NEW
```

to clear out any old variables and system control information so you can start a fresh program beginning at the new location. (If your computer is equipped with one or more disk drives, you can accomplish the same thing by loading a new program from a disk with the RUN command.)

RUN **command:** see Section 1.2.4

6.3 Shape Tables

Applesoft has a number of special facilities that allow you to manipulate shape tables defining shapes on the high-resolution graphics screen. Because shape tables have the advantages of both flexible design and very fast execution, they are ideal for applications such as on-screen animation. This section contains detailed information on creating and manipulating shape tables.

For Hackers Only: Since the advent of the Apple II series of computers, a number of excellent graphics software packages have appeared on the market. Available at most computer stores, these packages take the hard-core technical work (binary arithmetic and machine-language manipulation) out of designing and using shape tables. The information in this section is intended for those programmers who enjoy "twiddling the bits" themselves.

To use this section effectively, you'll need to know about bits and bytes and the rudiments of hexadecimal arithmetic. This information is available in any basic text on computer science; see the bibliography in the *Apple IIe Owner's Manual* that came with your computer. All computer memory addresses in this section are in hexadecimal; all other numbers, unless otherwise noted, are in decimal.

6.3.1 Creating a Shape Table

An Applesoft *shape definition* consists of a sequence of *plotting vectors* that are stored in a series of consecutive bytes in the computer's memory. One or more such shape definitions, with their indices (see "The Shape Table Index," below), make up a *shape table.*

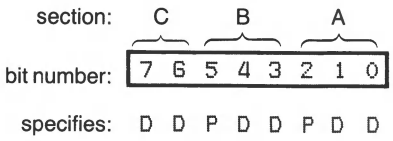
shape table: a collection of one or more shape definitions, together with their indices

Plotting Vectors

Each byte in a shape definition has three sections. Each section may contain a *plotting vector,* specifying whether to plot a point at the current screen position and in what direction to move (up, down, left, or right) before processing the next vector. Thus each byte can represent up to three plotting vectors.

plotting vector: a code representing a single step in drawing a shape on the screen

Figure 6-4 Plotting Vectors in a Byte



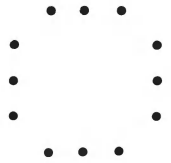
If DD = 00 move up
= 01 move right
= 10 move down
= 11 move left

If P = 0 don't plot
= 1 plot a point

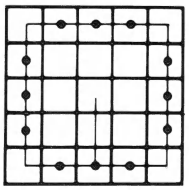
DRAW **and** XDRAW **statements:** see Section 6.3.2

Figure 6-5 Plotting a Shape

a.



b.



c.

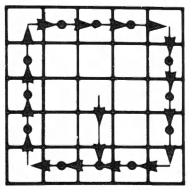


Figure 6-4 shows how the three sections are arranged within each of the bytes that make up a shape definition. In each plotting vector, bit P specifies whether to plot a point before moving, and the pair of bits designated DD specify the direction in which to move before processing the next vector.

Notice that the last section in each byte (the two high-order bits, labeled C in the figure) does not include a P bit. The value of P in such a section is always assumed to be 0 (don't plot); thus section C can only specify a move without plotting.

How Plotting Vectors Are Interpreted

The DRAW and XDRAW statements read through each byte in the shape definition, from the first byte in the definition to the last. Within each byte, the sections are processed from right to left: section A, then B, then C. When a byte is encountered that contains all zeros, the shape definition is complete.

At any section in the byte, if all the remaining sections contain only zeros, then those sections are ignored. Thus a byte can't end with a move in section C of 00 (move up without plotting), because that section, containing only zeros, will be ignored. Similarly, if section C is 00 (ignored), then section B cannot be a move of 000 (move up without plotting), since that will also be ignored. And a vector of 000 in section A will end your shape definition unless there is a one bit somewhere in section B or C.

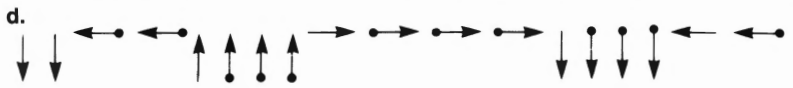
Coding a Shape Table

Suppose you want to draw a shape like that shown in Figure 6-5a. To convert the shape into an Applesoft shape definition, follow these steps:

1. Draw the shape on graph paper, one dot per square.
2. Decide where to start drawing the shape—let's start this one at the center—and draw a path through each point in the shape, using only 90-degree angles on the turns, as in Figure 6-5b.
3. Redraw the shape as a series of plotting vectors, each vector moving one place up, down, left, or right, and distinguish those vectors that plot a point before moving. This step is illustrated in Figure 6-5c; vectors that plot before moving are marked in the figure with a dot at the beginning of the direction arrow.

4. "Unwrap" the vectors and write them out in sequential order, as in Figure 6-5d.

Figure 6-5 continued



Now you're ready to code the plotting vectors as a shape definition table. Figure 6-6 gives the binary codes corresponding to each possible vector. For each vector in the shape, determine the proper bit code and place it in the next available section in the table, as shown in Figure 6-7. If the code won't fit (for instance, the vector in section C can't plot a point) or is a 00 (or 000) at the end of a byte, then just fill that section with zeros.

Figure 6-6 Codes for Plotting Vectors

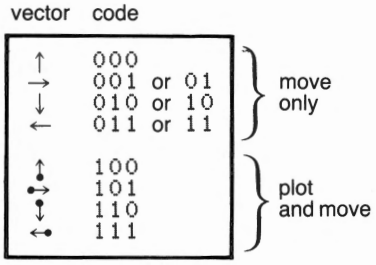


Figure 6-7 Shape Definition Table

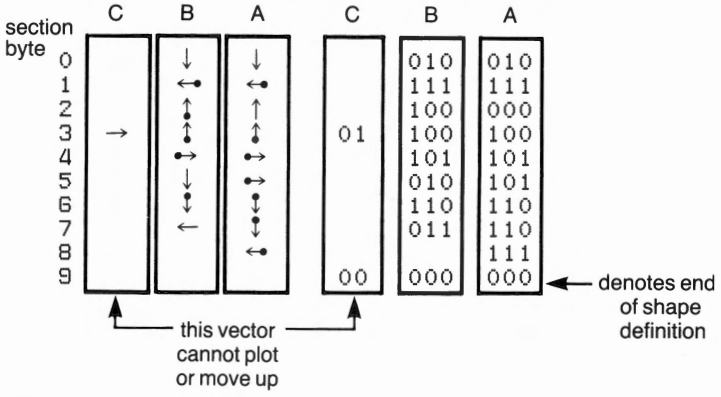


Table 6-3 Hexadecimal Byte Codes

Binary		Hex
0000	=	0
0001	=	1
0010	=	2
0011	=	3
0100	=	4
0101	=	5
0110	=	6
0111	=	7
1000	=	8
1001	=	9
1010	=	A
1011	=	B
1100	=	C
1101	=	D
1110	=	E
1111	=	F

The final step is to convert the binary codes representing the plotting vectors into hexadecimal form so you can type them into the computer. As shown in Table 6-3, each hexadecimal code corresponds to a group of four bits; so each row of eight bits in your definition table is represented by two such codes (called hexadecimal digits). This step is illustrated in Figure 6-8. The final shape definition for the shape in Figure 6-5 is

```
12 3F 20 64 2D 15 36 1E 07 00
```

The Shape Table Index

There is still a little more information you need to provide before you have a complete shape table: the table must have an *index.* This is simply a list indicating where in memory to find a particular shape. Applesoft needs the index so that it can find the shape later, when your program tries to draw the shape on the screen.

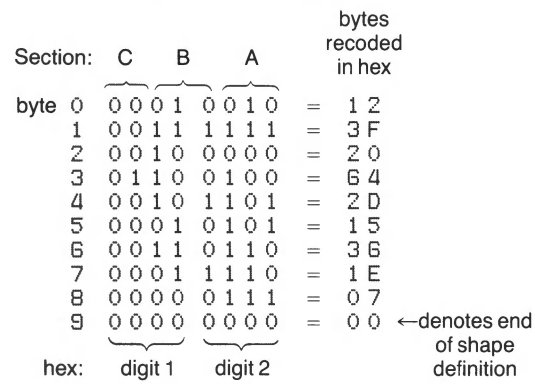


Figure 6-8 Converting the Shape Definition to Hexadecimal

The form of a complete shape table, including the index, is shown in Figure 6-9. The shape table's starting location, whose address is called S in the figure, contains the number of shape definitions in the table (between 00 and FF) in hexadecimal. The next byte (address S + 1) is unused; it is followed by a sequence of two-byte pairs giving the locations of the shapes in the table. (Notice that the shape locations are given with the bytes reversed—low-order byte first—and that the locations are specified relative to address S, the start of the table itself, and not in absolute memory addresses.) For simplicity, the shape definitions themselves are usually placed immediately after the index.

Figure 6-9 Form of a Complete Shape Table

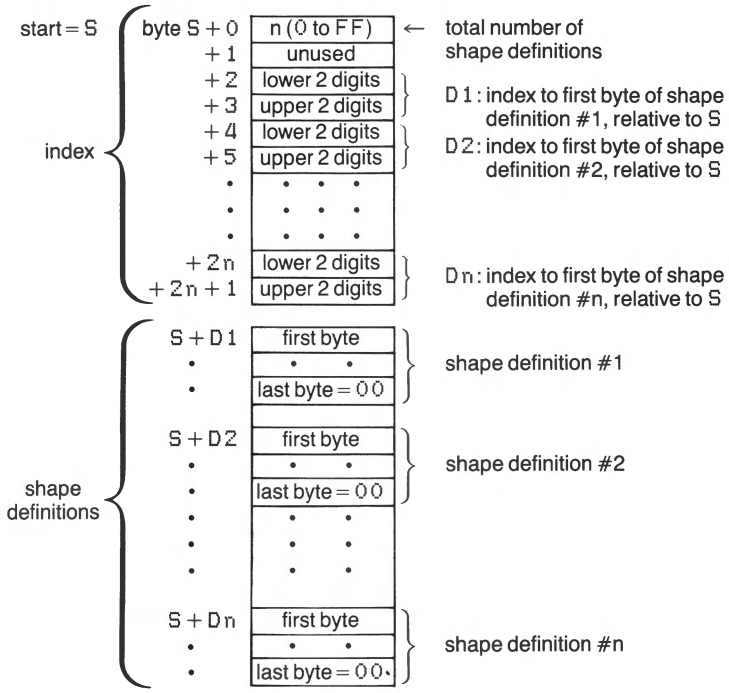


Figure 6-10 A Complete Shape Table

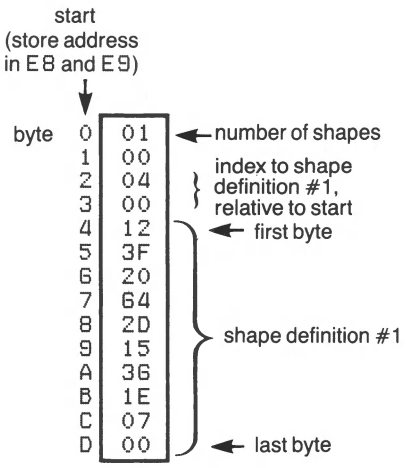


Figure 6-10 shows the complete shape table for our example. Since there's only one shape in the table, location S contains the value 1. Bytes S + 2 and S + 3 are needed to specify the shape's location; the shape definition itself can start in the next available byte, S + 4. So index byte S + 2 contains the value 04 and index byte S + 3 contains the value 00. Next come the bytes of the shape definition, as derived in Figure 6-8. The table ends with the zero byte marking the end of the shape definition.

Loading a Shape Table into Memory

Now that you've figured out how to code your shape in the form of a shape table, you have to get it into the computer's memory so Applesoft can draw it on the screen. You also have to tell Applesoft where in memory to look for the shape table.

First you must choose a starting address. This address must be less than the highest memory address available in your system, and must not be located in the high-resolution graphics page that you'll be using to display the shapes (locations 2000 through 3FFF for page 1, 4000 through 5FFF for page 2). For this example, we'll use hexadecimal address 1DFC, which is just below high-resolution page 1.

high-resolution graphics pages: see Sections 6.2.1, 6.2.2

Keep shape table out of harm's way

Warning

Be sure you don't place your shape table in an area that will conflict with your program or variable space, or with vital internal information used by the system. See the box labeled "Protecting Your Shape Table," below, for information on how to keep your program and shape table out of each other's way; see Appendix H for the memory locations of important system information.

Apple IIe Monitor program: see *Apple IIe Reference Manual*

DRAW **statement:** see Section 6.3.2

While you're in the process of creating the shape table, you'll probably want to type the table into memory directly from the keyboard using the Monitor program. Then you can draw the shape on the screen with an immediate-execution DRAW statement, see if it looks the way you want it, and go back and change it if it doesn't. See your *Apple IIe Reference Manual* for information on the use of the Monitor program.

POKE **statement:** see Section 7.1.2

Once your shape table looks correct, you'll want to be able to use it from within a program. Your program can store the table into memory by using POKE. To do this, you have to convert the starting address of the table, and each byte of the table itself, from hexadecimal to decimal, then store the decimal values into memory one at a time.

The shape table we've been developing consists of the hexadecimal bytes

```
01 00 04 00 12 3F 20 64 2D 15 36 1E 07
   00
```

The equivalent decimal values are

```
1 0 4 0 18 63 32 100 45 21 54 30 7 0
```

The starting address we've chosen for the table, hexadecimal 1DFC, is equivalent to 7676 decimal. So the following statements in a program will store the shape table into memory:

```
10 FOR X = 7676 to 7689
                          —memory locations where
                            shape table will go
20 READ A                 —read byte of table
30 POKE X, A              —store at next location
40 NEXT X                 —go back for next byte
50 DATA 1,0,4,0,18,63,32,100,45,21
     54,30,7,0            —contents of table
```

Another way for a program to store a shape table into memory is to load it from a disk or tape cassette. Details are given below under "Saving and Loading a Shape Table."

Store the starting address of the shape table

Now that you have your shape table in memory, you have to tell Applesoft where to find it. Applesoft looks for the table's starting address in hexadecimal locations E8 (low-order byte) and E9 (high-order byte), so you have to arrange somehow to store the correct starting address into these locations. If you've been using the Monitor program to type the shape table into memory from the keyboard, you can type its address into locations E8 and E9 in the same way. From within a program, you can do it with two more POKE statements. The hexadecimal addresses E8 and E9 are equivalent to decimal 232 and 233; the two bytes of the table's starting address, 1D and FC, are equivalent to 29 and 252. So the following POKE statements will do the trick:

```
60 POKE 232, 252 : POKE 233, 29
```

Your shape table is now stored correctly in the computer's memory, ready to be drawn on the screen from within your program with a DRAW statement.

Remember to store the two bytes of the starting address in reverse order, with the low-order byte before the high-order byte. This convention is always followed when storing memory addresses in the Apple IIe's memory.

SHLOAD **statement:** see Section 6.3.2

When you use SHLOAD to load a shape table from a tape cassette, the starting address is set up for you automatically in the proper locations.

Protect your tables

HIMEM: **statement:** see Section 7.2.1

Protecting Your Shape Table: In choosing a location in memory for your shape table, it's important to keep it out of the way of your Applesoft program, so the two don't overwrite each other. One way to do this is simply to use HIMEM: to set the upper limit of program memory to the starting address of the table. In the example,

```
HIMEM: 7676
```

This too is done automatically when you use SHLOAD to get the table from a tape cassette.

Unfortunately, this method leaves very little room for your program and variables—in the example, only 5628 bytes (7676 minus 2048). You can buy a little more space for your program by setting HIMEM: to the

beginning of the graphics page you're using (8192 for page 1, 16384 for page 2), as suggested in Section 6.2.5, "Protecting High-Resolution Graphics." You can then locate the shape table above the graphics page: that is, above location 16384 if you're using page 1, 24576 if you're using page 2.

Perhaps the best method is to locate the program and variables above the graphics page, again as described in Section 6.2.5. This leaves room for the shape table below the start of the graphics page. If you're using graphics page 1, that's 6144 bytes (8192 minus 2048)—enough room for a very extensive shape table!

Don't overwrite DOS!

Warning

If you locate your shape table above the high-resolution graphics page and your system is equipped with one or more disk drives, be careful not to run into the memory space occupied by the Disk Operating System, beginning at location 38400 (hexadecimal 9600).

Saving and Loading a Shape Table

Saving a shape table on a disk

To save your shape table on a disk, you need to know two things:

- The starting address of the table (1DFC in the example)
- The length of the table in bytes (14 in the example—hexadecimal 000E—including the "stop" byte)

Next, you must choose a file name under which to store your shape table on the disk. We'll use SHAPE1 for this example.

DOS: Disk Operating System

To save the table on a disk in immediate execution, put the disk in the disk drive and issue the following DOS command:

BSAVE **command:** see DOS manual

```
BSAVE SHAPE1, A$1DFC, L$000E
```

binary file: a file containing "raw" information not expressed in text form

This command says "store a binary file named SHAPE1 on the disk, containing the current contents of memory starting at hexadecimal address 1DFC, and 000E (hexadecimal) bytes long."

If you're using a disk drive other than the main startup drive, the BSAVE command should also include slot and drive parameters specifying which disk drive to use; see your DOS manual for details.

To issue the same command from within an Applesoft program, use the statement

```
PRINT CHR$ (4); "BSAVE SHAPE1, A$1DFC,
   L$000E"
```

Again, see your DOS manual for details.

Loading a shape table from a disk

To load the table back into memory from the disk, you can use the DOS command

BLOAD **command:** see DOS manual

```
BLOAD SHAPE1
```

in immediate execution, or the statement

```
PRINT CHR$ (4); "BLOAD SHAPE1"
```

from within a program. Notice that you don't have to include the starting address and the table length; this information will be picked up automatically from within the disk file itself. However, the starting address is not stored automatically into the special addresses where Applesoft looks for them, so you (or your program) will have to do that for yourself:

Don't forget the starting address

```
POKE 232, 252 : POKE 233, 29
```

Saving a shape table on tape

To save your shape table on a tape cassette, you need to know three things:

- The starting address of the table (1DFC in the example)
- The last address of the table (1E09 in the example)
- The difference between the first two items (hexadecimal 000D, decimal 14)

Item 3, the difference between the last address and the first address of the table, must be stored in locations 0 (low-order byte) and 1 (high-order byte). From the Monitor, type

Apple IIe Monitor program: see *Apple IIe Reference Manual*

```
0:0D 00
```

and press RETURN. Now you must write to the cassette first the table length from locations 0 to 1, then the shape table itself:

```
0.1W 1DFC.1E09W
```

> Don't press the RETURN key until you've put a cassette in your tape recorder, rewound it, and started it recording.

Loading a shape table from tape

To load the shape table back from the tape, use the SHLOAD statement.

SHLOAD **statement:** see Section 6.3.2

6.3.2 *Using Shape Tables*

The commands in this section are used to draw and manipulate on the screen shapes defined by a shape table in memory:

- DRAW and XDRAW draw shapes from a shape table onto the high-resolution screen.
- SCALE= controls the scale at which shapes are drawn on the screen.
- ROT= controls the rotation of shapes on the screen.
- SHLOAD loads a shape table into memory from a tape cassette.

As a preview of what the commands in this section can do, here's a sample Applesoft program for you to try. The program first stores into memory the shape table developed in Section 6.3.1, using the POKE statement (see "Loading a Shape Table into Memory)." Then it uses the statements described in this section to produce a somewhat surprising effect on the screen. See if you can guess what the program will display, then type it and run it:

```
 10 FOR X = 7676 TO 7689
                              —memory locations where
                                shape table will go
 20 READ A                    —read byte of table
 30 POKE X, A                 —store at next location
 40 NEXT X                    —go back for next byte
 50 DATA 1,0,4,0,18,63,32,100,45,21,
      54,30,7,0               —contents of table
100 HGR2
110 HCOLOR= 3
120 FOR R = 1 TO 50
130 ROT= R
140 SCALE= R
150 XDRAW 1 AT 140, 96
160 NEXT R
170 GOTO 120
```

When you get tired of watching the show, interrupt the program by pressing CONTROL-C to regain control of the system.

CONTROL-C: see Section 1.3.2

The DRAW *Statement*

```
DRAW 5
DRAW 1 AT 140, 96
DRAW SHAPE AT XCENTER + XOFFSET,
   YCENTER + YOFFSET
```

DRAW draws a shape on the high-resolution screen

The DRAW statement draws a shape from a shape table on the high-resolution graphics screen at a specified location. The expression following the keyword DRAW gives the index number of the desired shape within the shape table currently in memory. The location at which the shape is to be drawn is specified by a pair of expressions following the keyword AT, separated by a comma. The first expression gives the horizontal screen position of the shape's starting point; the second gives the vertical position.

shape tables: see Section 6.3.1

HCOLOR = **statement:** see Section 6.2.3

The designated shape is drawn in the current display color, scale, and rotation, as specified in the most recently executed HCOLOR =, SCALE =, and ROT = statements.

Warning

You must specify the color, scale, and rotation of the shape before DRAW is executed. If any of these have not been specified, the results will be random: odd dots may appear, bizarre shapes may be drawn, and memory may be overwritten.

Assuming that a shape table is already loaded into memory (see "Loading a Shape Table into Memory" in Section 6.3.1), the following program will draw the first shape in the table at column 50, row 100:

```
10 HGR                          — display high-resolution
                                  graphics
20 HCOLOR= 3                    —set color to white-1
30 ROT= 0                       —orient shape as originally
                                  defined
40 SCALE= 5                     —enlarge shape 5 times
50 DRAW 1 AT 50, 100            —draw shape 1 at column 50,
                                  row 100
```

If you omit the keyword AT and the screen coordinates,

```
50 DRAW 1
```

HPLOT **statement:** see Section 6.2.4

XDRAW **statement:** see below

Applesoft will put the shape on the screen starting at the last point plotted by the most recently executed HPLOT, DRAW, or XDRAW statement. (The shape drawn on the screen may not actually begin at the last point previously plotted. If the first plotting vector in the shape doesn't actually plot a point, there will be an offset between the first visible point in the shape and the last point plotted.) If no such statement has been executed, the results are unpredictable.

If the shape number specified is less than 0 or greater than the actual number of shapes in the shape table, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

Be sure to load a shape table first

CONTROL-RESET: see Section 1.3.2

HGR **statement:** see Section 6.2.1

Warning

If you execute DRAW without first loading a shape table into memory, the system may hang (use CONTROL-RESET to recover), or Applesoft may draw random shapes anywhere in the high-resolution graphics area of memory (locations 8192 to 24575 decimal), whether or not HGR or HGR2 has previously been executed. This can have disastrous consequences if your program is longer than about 6000 bytes.

The XDRAW *Statement*

```
XDRAW 5
XDRAW 1 AT 140, 80
XDRAW SHAPE AT XCENTER + XOFFSET,
   YCENTER + YOFFSET
```

XDRAW **erases a shape**

The XDRAW statement works exactly the same as DRAW, except that the color used to draw the shape is the complement of the color already existing at each point plotted. The following pairs of colors are complements:

Complementary colors

- black and white
- violet and green
- blue and orange

XDRAW is most commonly used to erase a previously drawn shape. The following program, which assumes that a shape table has already been loaded into memory (see "Loading a Shape Table into

Memory" in Section 6.3.1), illustrates the point by drawing and then erasing the same shape, leaving the screen blank:

```
10 HGR2                          —display full-screen high-reso-
                                  lution graphics
20 HCOLOR= 3                     —set color to white-1
30 ROT= 0                        —orient shape as originally
                                  defined
40 SCALE= 5                      —enlarge shape 5 times
50 DRAW 1 AT 50, 100             —draw shape at column 150,
                                  row 100
60 FOR Z = 1 TO 500 : NEXT Z
                                 —stall for a short time
70 XDRAW 1 AT 50, 100            —erase the shape
```

If you use DRAW and XDRAW alternately in a loop, you can do animation:

```
10 HGR2                          —display full-screen high-resolu-
                                  tion graphics
20 HCOLOR= 4                     —set color to white-1
30 ROT= 0                        —orient shape as originally
                                  defined
40 SCALE= 5                      —enlarge shape 5 times
50 FOR X = 1 TO 200              —loop 200 times
60 DRAW 1 AT 50 + X, 100
                                 —draw shape in a different col-
                                  umn each time
70 XDRAW 1 AT 50 + X, 100
                                 — erase shape
80 NEXT X                        —repeat loop
```

If the shape number specified is less than 0 or greater than the actual number of shapes in the shape table, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

Be sure to load a shape table first

CONTROL - RESET : see Section 1.3.2

HGR **statement:** see Section 6.2.1

Warning

If you execute XDRAW without first loading a shape table into memory, the system may hang (use CONTROL - RESET to recover), or Applesoft may draw random shapes anywhere in the high-resolution graphics area of memory (locations 8192 to 24575 decimal), whether or not HGR or HGR2 has previously been executed. This can have disastrous consequences if your program is longer than about 6000 bytes.

The SCALE= *Statement*

```
SCALE= 10
SCALE= Z / 4
```

SCALE= sets scale factor for DRAW and XDRAW

DRAW and XDRAW statements: see above

The SCALE= statement sets the scale factor (relative size) for the high-resolution graphics shape to be drawn by DRAW or XDRAW. The expression following the keyword SCALE= specifies the scale factor, which may range from 1 (reproduce the shape exactly as originally defined) up to a maximum of 255 (draw the shape 255 times the size originally defined).

Assuming that a shape table is already loaded into memory (see "Loading a Shape Table into Memory" in Section 6.3.1), the following program will draw the first shape in the table at three different positions on the screen and in three different sizes:

```
10 HGR2                          —display full-screen high-reso-
                                   lution graphics
20 HCOLOR= 3                     —set color to white-1
30 ROT= 0                        —orient shape as defined
40 SCALE= 1                      —use original size
50 DRAW 1 AT 100, 100            —draw shape at column 100,
                                   row 100
60 SCALE= 2                      —scale to twice original size
70 DRAW 1 AT 150, 100            —draw at column 150, row 100
80 SCALE= 3                      —scale to three times size
90 DRAW 1 AT 200, 100            —draw at column 200, row 100
```

A scale setting of 0 is considered equivalent to the maximum setting (255). If the scale setting specified is less than 0 or greater than 255, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

Scale factors are useful only up to a certain point. Large scale settings produce some rather outlandish results on the screen.

Notice that the equal sign is part of the keyword SCALE=; it doesn't represent an assignment to a variable named SCALE. A statement such as

```
LET SCALE= X
```

will cause a syntax error. The only way to find out the current scale setting is to keep track of it yourself with a separate variable.

The ROT= *Statement*

```
ROT= 16
ROT= 32 + 2 * R
```

ROT= **sets rotation for** DRAW **and** XDRAW

DRAW **and** XDRAW **statements:** see above

The ROT= (for "rotation") statement sets the angular rotation for the high-resolution graphics shape to be drawn by DRAW or XDRAW. The expression following the keyword ROT= specifies the rotation in units of 5.625 degrees (1/64 of a circle). ROT= 0 will orient the shape exactly as defined in the shape table, ROT= 16 will rotate the shape 90 degrees clockwise, ROT= 32 will rotate it 180 degrees, and so on. The process repeats starting at ROT= 64.

Assuming that a shape table is already loaded into memory (see "Loading a Shape Table into Memory" in Section 6.3.1), the following program will draw the first shape in the table, five times its original size, at two different positions on the screen, once oriented as originally defined and once rotated by 45 degrees:

```
10 HGR2                          —Display full-screen high-resolution graphics
20 HCOLOR= 3                     —set color to white-1
30 SCALE= 5                      —scale shape to five times original size
40 ROT= 0                        —orient shape as originally defined
50 DRAW 1 AT 50, 100             —draw shape at column 50, row 100
60 ROT= 8                        —rotate shape 45 degrees
70 DRAW 1 AT 100, 100            —draw shape at column 100, row 100
```

Rotation also depends on scale setting

scale setting: see above

The amount of rotation obtainable is somewhat dependent on the current scale setting. For SCALE= 1, Applesoft recognizes only four rotation values (0, 16, 32, 48); for SCALE= 2, it recognizes eight rotation values (0, 8, 16, . . .); for SCALE= 3, it recognizes twelve rotation values; and so on. For scale settings of 16 or more, the full range of rotation values is available. For unrecognized rotation values, Applesoft usually orients the shape with the next smallest rotation that it recognizes.

Notice that the equal sign is part of the keyword ROT=; it doesn't represent an assignment to a variable named ROT. A statement such as

```
LET ROT= X
```

will cause a syntax error. The only way to find out the current rotation setting is to keep track of it yourself with a separate variable.

If the rotation setting specified is less than 0 or greater than 255, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

The SHLOAD *Statement*

```
SHLOAD
```

SHLOAD **loads a shape table from tape**

HIMEM: **statement:** see Section 7.2.1

The SHLOAD statement (for "shape load") loads a shape table from a tape cassette. The shape table is loaded just below the upper limit of available program and variable space (HIMEM:); HIMEM: is then set just below the shape table to protect it.

To use SHLOAD in immediate execution, turn on your tape recorder with the proper tape inserted and cued up to the proper place. Then type

```
SHLOAD
```

and press RETURN. You should hear one "beep" when the shape table's length has been read successfully, and another when the table itself has been read.

You can also use SHLOAD from within a program (with appropriate prompting messages) to allow users to load their own shape tables:

```
100 PRINT "CUE UP YOUR SHAPE TAPE AND
      PRESS THE PLAY BUTTON."
110 PRINT "THEN PRESS THE RETURN KEY TO
      LOAD THE SHAPE TABLE."
                          —prompt user with instructions
120 GET STALL$            —wait for keypress
130 SHLOAD                —load shape table from tape
140 PRINT "TABLE LOADED—PLEASE SHUT OFF
      YOUR RECORDER." —tell user table is loaded
```

If you load a second shape table replacing the first one, you or your program should reset HIMEM: to avoid wasting memory. See the section "Loading a Shape Table into Memory" in Section 6.3.1 for more information on shape tables and memory usage.

Don't forget to turn on your tape recorder!

CONTROL - RESET : see Section 1.3.2

If you try to use SHLOAD without a tape recorder connected, turned on, and set to play, the system will hang indefinitely. Use CONTROL - RESET to regain control.

If a variable name begins with the reserved word SHLOAD

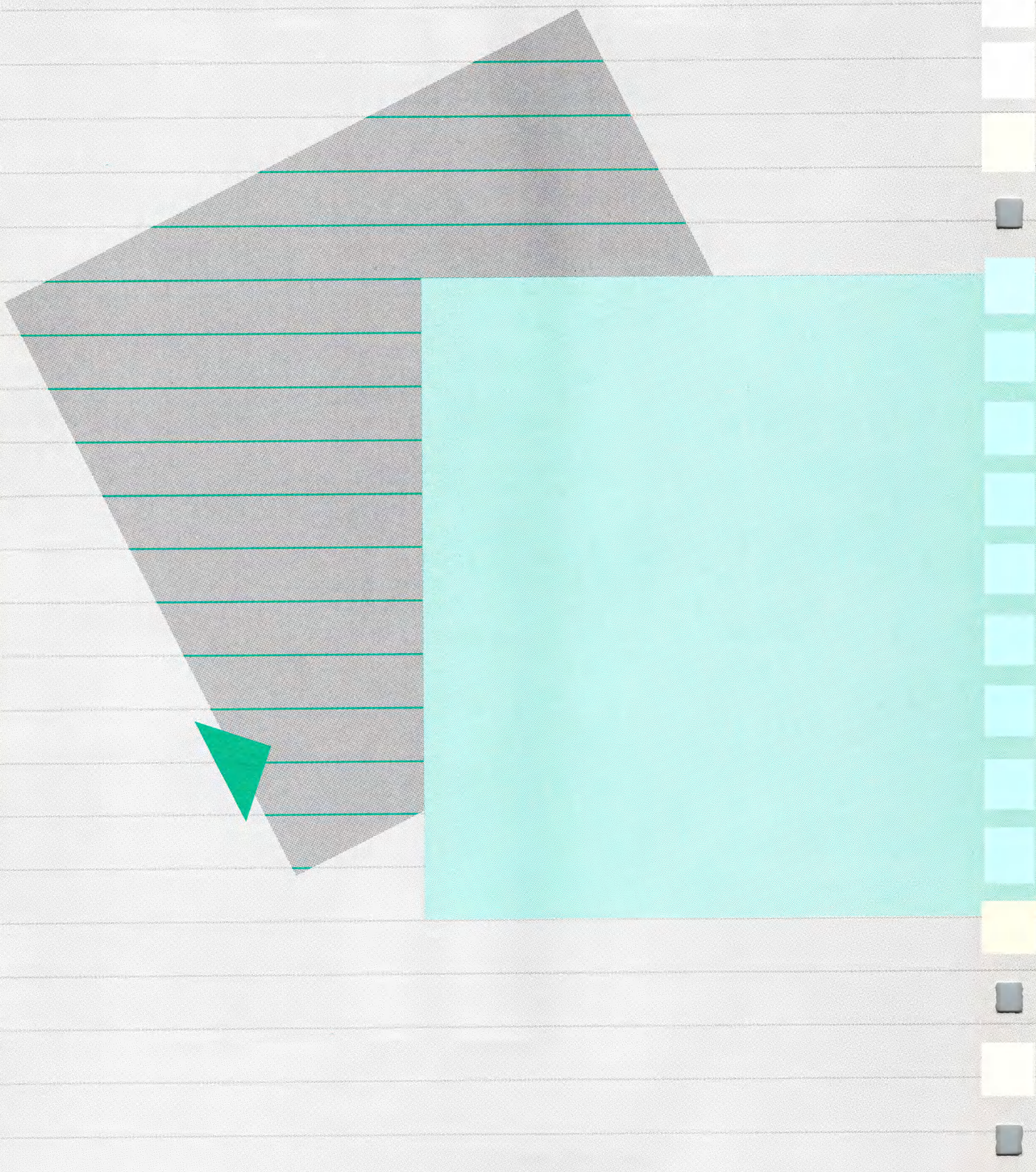
```
SHLOADER = 59
```

SHLOAD may be executed before a syntax error is detected. In such a case, the system will patiently wait (forever, if necessary) for a shape table to be loaded from a tape cassette. Again, use CONTROL - RESET to recover.

For more information...

For information on saving a shape table on tape, see "Saving and Loading a Shape Table" in Section 6.3.1. See Appendix M for a list of all statements dealing with tape cassettes.

Utility Statements



Utility Statements

The features covered in this chapter are concerned with low-level control of the programming environment.

direct memory access: see Section 7.1

Section 7.1, "System Utilities," discusses direct access to specific locations in the computer's memory from within an Applesoft program.

memory management: see Section 7.2

Section 7.2, "Memory Management," describes the ways in which Applesoft programs can control the limits of program space.

debugging: see Section 7.3

Section 7.3, "Debugging," tells how to trace the execution of a running program for debugging purposes.

7.1 System Utilities

This section describes statements and functions that give Applesoft programs direct access to the Apple IIe's memory:

PEEK **function:** see Section 7.1.1

POKE **statement:** see Section 7.1.2

CALL **statement:** see Section 7.1.3

USR **function:** see Section 7.1.4

WAIT **statement:** see Section 7.1.5

- PEEK examines the contents of a memory location.
- POKE alters the contents of a memory location.
- CALL and USR allow Applesoft programs to execute machine-language subroutines stored in the computer's memory.
- WAIT suspends program execution until a specified signal is received from a peripheral device.

7.1.1 *The* PEEK *Function*

PEEK examines contents of a memory location

The PEEK function directly examines the contents of a specified location in the computer's memory. The argument given to PEEK is the decimal address of the desired memory location. PEEK yields the contents of the specified location, which will be an integer from 0 to 255. For example, the following program displays the contents of addresses 100 through 120:

```
10 FOR ADDR = 100
     TO 120                    —loop through desired
                                 addresses
20 PRINT "LOCATION "; ADDR; "HOLDS THE
     VALUE "; PEEK (ADDR) 30 NEXT ADDR
                               —display contents of location
                               —go back for next address
```

Certain locations hold **special information** or produce **special effects:** see Appendix F

Certain locations in the Apple IIe's memory hold special system information or produce special effects whenever their contents are read. One important use of PEEK is for manipulating these special locations. See Appendix F, "Peeks, Pokes, and Calls," for details.

If PEEK is given a negative argument value, it adds 65536 (2 to the 16th power) to obtain an equivalent positive address. For example,

```
PEEK (-16384)  is equivalent to  PEEK (49152)
PEEK (-1)      is equivalent to  PEEK (65535)
PEEK (-32768)  is equivalent to  PEEK (32768)
PEEK (-65500)  is equivalent to  PEEK (36)
```

If the argument is not in the range -65535 to +65535, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

7.1.2 *The* POKE *Statement*

```
POKE 34, 8
POKE -16302, 0
POKE ADDR, (2*D1 + 3*D2) / (U - V)
```

POKE alters contents of a memory location

The POKE statement stores a specified value directly into a location in the computer's memory. The first expression following the keyword POKE gives the decimal address of the memory location; the second

expression, separated from the first by a comma, gives the value to be stored into that location. For example,

```
POKE 34, 8          —stores value 8 into location
                     34
```

Warning

Be certain that the address into which you are storing doesn't contain part of your program or some vital system information that you don't want to change. An ill-advised POKE can hang the system, drop you into the Monitor, or alter the operation of the system or of your program in unpredictable and possibly disastrous ways. In the event of catastrophe, use CONTROL-RESET to regain control of the system. See Appendix H for the locations of vital system information that shouldn't be tampered with.

CONTROL-RESET: see Section 1.3.2

Certain locations in the Apple IIe's memory hold special system information or produce special effects whenever a value is stored into them. One important use of POKE is for manipulating these special locations. See Appendix F, "Peeks, Pokes, and Calls," for details.

If POKE is given a negative target address, it adds 65536 (2 to the 16th power) to obtain an equivalent positive address. For example,

```
POKE -16384, 0     is equivalent to  POKE 49152, 0
POKE -32768, 112   is equivalent to  POKE 32768,
                                     112
POKE -65502, 8     is equivalent to  POKE 34, 8
```

If the target address is not in the range -65535 to +65535, or if the specified value is not in the range 0 to 255, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

7.1.3 *The* CALL *Statement*

```
CALL 54915
CALL -936
CALL ROUTINE (J)
```

CALL executes a machine-language subroutine

The CALL statement executes a machine-language subroutine from within an Applesoft program. The decimal address of the desired subroutine follows the keyword CALL. Control is transferred to the

subroutine at the designated address; when the subroutine is finished, execution continues with the statement following the CALL. For example,

```
CALL 64668
```
— executes machine-language subroutine beginning at address 64668

Warning

Make sure the address you give in the CALL statement is the beginning of a valid machine-language subroutine! A misdirected CALL can have unpredictable and probably unpleasant consequences, such as hanging the system or dropping you into the Monitor. If any of these calamities befall you, use CONTROL-RESET to regain control of the system.

CONTROL-RESET: see Section 1.3.2

system calls: see Appendix F

The Apple IIe's built-in firmware contains many useful subroutines accessible with the CALL statement; see Appendix F, "Peeks, Pokes, and Calls," for details.

POKE **statement:** see Section 7.1.2

Apple IIe Monitor program: see *Apple IIe Reference Manual*

You can also use CALL to execute machine-language subroutines of your own, which you have stored into memory with the POKE statement, typed from the keyboard via the Monitor, or loaded into the computer from a disk or tape.

If CALL is given a negative target address, it adds 65536 (2 to the 16th power) to obtain an equivalent positive address. For example,

```
CALL -936   is equivalent to  CALL 64600
CALL -868   is equivalent to  CALL 64668
CALL -1998  is equivalent to  CALL 63538
```

If the target address is not in the range -65535 to +65535, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

7.1.4 *The USR Function*

Not for Everyone: This feature is intended for expert programmers only, and requires a knowledge of machine-language programming. Readers with fewer than sixteen fingers are advised to skip this section.

USR executes a machine-language function routine

The USR (for "user-supplied routine") function executes a machine-language function routine stored into the computer's memory by you, the user. Such a routine typically performs some high-speed computation that cannot be done efficiently, or cannot be expressed at all, in

Applesoft. The argument supplied to the USR function is passed unchanged to the machine-language routine, and the result yielded by the routine is passed back as the value of the USR call.

POKE statement: see Section 7.1.2

Apple IIe Monitor program: see *Apple IIe Reference Manual*

The function routine to be executed with USR may be stored into the computer's memory with the POKE statement, typed from the keyboard via the Monitor, or loaded into the computer from a disk or tape. When USR is called, the value supplied as an argument is placed into the floating-point accumulator in the computer's memory (hexadecimal locations $9D to $A3); control is then transferred via a machine-language JSR (Jump to Subroutine) instruction to hexadecimal address $0A (decimal 10). Locations $0A to $0C (decimal 10 to 12) must contain a machine-language JMP (Jump) instruction to the beginning of the machine-language routine. The routine should leave its result in the floating-point accumulator and return control to Applesoft with an RTS (Return from Subroutine) instruction. The contents of the floating-point accumulator are then passed back to your Applesoft program as the value yielded by USR.

Argument and result passed via **floating-point accumulator**

Here is a trivial example showing the use of the USR function. The machine-language routine shown here simply takes the argument value it receives and multiplies it by 8:

```
] CALL -151                    —leave Applesoft; enter Monitor
* 0A:4C 00 03                  —set up machine-language
                                 jump to hexadecimal address
                                 $300
* 0300:18 A5 9D 69 03 85 9D 60
                                —enter short machine-language
                                 routine to multiply contents of
                                 floating-point accumulator by
                                 8
* [CONTROL]-C                  —return to Applesoft
] PRINT USR (3)                —execute routine with argument
                                 value 3
24                             —result displayed on screen
```

Locations $0A to $0C must contain a JMP to the routine

At hexadecimal address $0A, there is a JMP (op code 4C) to hexadecimal address $300. (As usual in 6502 machine language, the low-order byte of the address, 00, precedes the high-order byte, 03.) Beginning at address $300 is a machine-language routine to multiply the value in the floating-point accumulator by 8. Back in Applesoft, when the function call USR (3) is executed, the argument value 3 is placed in the floating-point accumulator and control is passed to the machine-language routine via the JMP at location $0A. The machine-language routine gets the value in the floating-

point accumulator, multiplies it by 8, puts the result (24) back into the floating-point accumulator, and returns control to Applesoft with an RTS instruction (op code 60). The value 24 is then passed back as the result of the USR call.

To obtain a two-byte integer from the value in the floating-point accumulator, your machine-language routine should do a JSR to address $E01C. Upon return, the integer value will be in locations $A0 (high-order byte) and $A1 (low-order byte).

To convert an integer result to its floating-point equivalent, so that the function can return that value, place the two-byte integer in registers A (high-order byte) and Y (low-order byte). Then do a JSR to address $E2F2. Upon return, the floating-point value will be in the floating-point accumulator.

7.1.5 *The* WAIT *Statement*

```
WAIT 49347, 15
WAIT 49401, 240, 192
WAIT ADDR%, M1%, M2%
```

Novices need not apply

For Experts Only: This feature is intended for expert programmers only, and requires an understanding of bit-masking operations. If you think a mask is something you wear on Halloween, you can safely afford to skip this section. You won't miss a thing.

WAIT waits for a signal from a peripheral device

The WAIT statement suspends program execution until a specified bit pattern appears at a specified memory location. It is typically used to wait for a particular status signal from a peripheral device.

mask: a pattern of bits for use in bit-level logical operations

The first expression following the keyword WAIT designates the address of the memory location to be tested. The second expression represents a one-byte *mask* specifying which bits of the designated location are of interest: a one bit in the mask means that the corresponding bit of the memory location is to be tested; a zero bit means it is to be ignored. The optional third expression is another one-byte mask specifying the bit value to be tested for in each position of the memory location: a one bit in the mask tests for a zero bit in the corresponding position of the memory location, and vice-versa (!). If the second mask is omitted, all bit positions specified by the first mask will be tested for a one bit. For example,

```
WAIT ADDR, 255
```
—wait for a one bit anywhere in location ADDR

```
WAIT ADDR, 255, 255
```
—wait for a zero bit anywhere in location ADDR

```
WAIT ADDR, 1
```
—wait for low-order bit of location ADDR to become 1

```
WAIT ADDR, 128, 128
```
—wait for high-order bit of location ADDR to become 0

```
WAIT ADDR, 3, 2
```
—wait for low-order bit of location ADDR to become 1 or second low-order bit to become 0

When WAIT is executed, the contents of the location specified by the first expression are exclusive-or'ed with the mask represented by the third expression (if any); the result is then anded with the mask represented by the second expression. If the result is nonzero (that is, if any of the bits of interest are in the specified state), then program execution proceeds; if the result is zero (none of the bits of interest are in the specified state), then the test is repeated. Thus program execution will be suspended until one of the specified bits is set to the specified state by an outside agency (presumably a signal from a peripheral device).

Warning

If the specified bit pattern never appears, program execution will hang forever. Make sure that the memory location you're testing is receiving information that will eventually test true. The only way to interrupt a WAIT is with CONTROL - RESET .

CONTROL - RESET : see Section 1.3.2

If WAIT is given a negative target address, it adds 65536 (2 to the 16th power) to obtain an equivalent positive address. For example,

```
WAIT -16189, 15
```
is equivalent to
```
WAIT 49347, 15
```

If the target address is not in the range -65535 to +65535, or if either of the masks is not in the range 0 to 255, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

7.2 Memory Management

The features discussed in this section are used to control the way Applesoft allocates memory space for your program:

HIMEM: **statement:** see Section 7.2.1

LOMEM: **statement:** see Section 7.2.2

FRE **function:** see Section 7.2.3

- The HIMEM: statement sets the upper limit of available program memory.
- The LOMEM: statement sets the lower limit of available program memory.
- The FRE function determines the amount of remaining memory space available to the program.

7.2.1 The HIMEM: Statement

```
HIMEM: 8192
```

HIMEM: **sets upper limit of available program memory**

The HIMEM: statement sets the highest memory address available to an Applesoft program for storage of program lines and variables. The upper limit of available program memory is set to the value of the expression following the keyword HIMEM:. The area above this address is available for use by the Disk Operating System, high-resolution graphics, or machine-language programs.

Notice that the colon is part of the keyword HIMEM: and is required.

Applesoft automatically sets HIMEM: to the address of the highest writable memory (RAM) address available on your computer. On systems equipped with disk drives, loading the Disk Operating System (DOS) will automatically reset HIMEM: to a lower value in order to protect the area of memory occupied by DOS itself. See your DOS manual for further information.

Loading DOS resets HIMEM:

You can change the setting of HIMEM: only by

Apple IIe Monitor program: see *Apple IIe Reference Manual*

- executing the HIMEM: statement
- typing CONTROL-B to the Monitor program
- restarting the system
- loading a machine-language program

A word to the wise

Warning

Resetting HIMEM: above its current value is an extremely dangerous practice that can result in writing over the Disk Operating System or other vital system information. Wise programmers will carefully investigate reserved memory areas before writing to them.

A common use of HIMEM: is to protect your program and high-resolution graphics from overwriting each other. See Section 6.2.5, "Protecting High-Resolution Graphics," for details.

Helpful Hint: The current value of HIMEM: is stored in decimal memory locations 115 and 116; to obtain that value, use the expression

PEEK **function:** see Section 7.1.1

```
PEEK (116) * 256 + PEEK (115)
```

If HIMEM: is given a negative address, it adds 65536 (2 to the 16th power) to obtain an equivalent positive address. For example,

```
HIMEM: -57344
```
is equivalent to
```
HIMEM: 8192
```

If the specified address is not in the range -65535 to +65535, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

If the specified address is lower than the current setting of LOMEM:, or doesn't allow enough room for the program already in memory, the program will halt with the message

```
?OUT OF MEMORY ERROR
```

7.2.2 *The* LOMEM: *Statement*

```
LOMEM: 24576
```

LOMEM: **sets lower limit of available program memory**

The LOMEM: statement sets the lowest memory address available to an Applesoft program for storage of variables. The lower limit of available program memory is set to the value of the expression following the keyword LOMEM:. The area below this address is available for high-resolution graphics or machine-language programs. LOMEM: also resets all variables to their initial values and wipes out all functions defined with DEF FN.

DEF FN **statement:** see Section 2.4.3

Notice that the colon is part of the keyword LOMEM: and is required.

Adding a program line resets LOMEM:

Applesoft ordinarily begins to store variables at the end of the program in memory. Each time you add, delete, or change a program line, Applesoft resets LOMEM: to a location just above the final line of the program. Executing the NEW command or typing CONTROL-B to the Monitor resets LOMEM: to its initial value.

NEW **command:** see Section 1.2.1

The value of LOMEM: can only be increased from its current setting. An attempt to set LOMEM: to a lower value than the one already in effect will halt the program with the message

```
?OUT OF MEMORY ERROR
```

Don't execute LOMEM: from within a program

Warning
Changing LOMEM: during the course of a program is a most dangerous practice that can cause portions of the program or of Applesoft's internal control information to become unavailable, which in turn will cause the program to behave in outlandish ways (if at all). Programmers who behave with such reckless abandon have only themselves to blame.

Helpful Hint: The current value of LOMEM: is stored in decimal memory locations 105 and 106; to obtain that value, use the expression

PEEK **function:** see Section 7.1.1

```
PEEK (106) * 256 + PEEK (105)
```

If LOMEM: is given a negative address, it adds 65536 (2 to the 16th power) to obtain an equivalent positive address. For example,

```
LOMEM: -49152  is equivalent to  LOMEM: 16384
```

If the specified address is not in the range -65535 to +65535, the program will halt with the message

```
?ILLEGAL QUANTITY ERROR
```

HIMEM: **statement:** see Section 7.2.1

If the specified address is higher than the current setting of HIMEM:, or lower than the address of the highest memory location occupied by the current operating system (plus any currently stored program), the program will halt with the message

```
?OUT OF MEMORY ERROR
```

7.2.3 *The* FRE *Function*

FRE **yields amount of available memory**

The FRE function yields the number of bytes of unused writable (RAM) memory available to the running program. For example,

```
LET AVAIL = FRE(0)     —set AVAIL to amount of
                        available memory remaining
```

Notice that the name of the function is FRE, not FREE.

If the number of free bytes exceeds 32767, FRE yields a negative result; adding 65536 will give you the actual number of free bytes:

```
IF FRE(0) < 0 THEN AVAIL = FRE(0) +
   65536
```

HIMEM: **statement:** see Section 7.2.1

If you have set HIMEM: beyond the highest RAM address in your Apple IIe, FRE may yield a value higher than the computer's actual memory capacity. The reliability of such a value is to be taken lightly.

Argument required but ignored

Stranger Than Fiction: The argument given to FRE is ignored, and has no effect on the operation of the function. However, you can't leave it out—you must include an argument expression of some kind to "keep the parentheses apart." What you use for an argument expression doesn't matter, but if Applesoft can't evaluate it as a legal expression, you'll get an error halt.

How Applesoft Manages Free Space: The amount of free space reported by FRE is the number of bytes remaining below the string storage space and above the numeric array and string pointer array space (see Section H.2, "Applesoft Memory Allocation"). When Applesoft changes the contents of a string variable during the course of a program (say from "CAT" to "DOG"), the characters in the old string ("CAT") are not erased from memory; Applesoft simply allocates some fresh space to hold the new string ("DOG"). As a result, characters left over from unused strings take up "dead space" and slowly fill memory from HIMEM: down toward the top of the array space.

Applesoft will automatically clear out these leftover characters when the bottom of string space "collides" with the top of array space. But if you're using any of the free space for machine-language programs or for high-resolution graphics, they may be overwritten.

Light Housecleaning: The automatic "housecleaning" just described takes time (anywhere from a fraction of a second to over two minutes, depending on the number of string variables your program is using). Furthermore, such housecleaning occurs at unpredictable moments—whenever your string and array spaces happen to collide. If it happens while Applesoft is in the middle of displaying a message on the screen, for instance, it can cause unfortunate confusion for your program's user, who will be left waiting for the computer to finish displaying a half-delivered message.

The FRE function provides a tool for warning the user that the computer will be busy for a while. The address of the current beginning of string space is kept in locations 111 and 112 of the computer's memory; the end of array space is kept in locations 109 and 110. Whenever Applesoft needs to allocate more memory, it compares the contents of these locations; if they differ by less than one, Applesoft does its automatic housecleaning.

Since Applesoft checks these locations, so can you. When the difference between them starts getting close to zero, it's time to display some kind of "don't worry" message and force housecleaning. Using a statement of the form

```
IF (PEEK(112)*256 + PEEK(111))
  -(PEEK(110)*256 + PEEK(109)) > 2 THEN
  PRINT "PLEASE STAND BY..." : Q = FRE (0)
```

periodically within your program will force housecleaning to occur and will prevent such confusion.

Since the housecleaning can take as long as several minutes each time it occurs, don't do it too often. It's best to use `FRE (0)` when you need a pause anyway—such as after you write information onto a disk, or while the user is reading information on the display screen.

7.3 Debugging Facilities

This section details two Applesoft commands used as debugging aids: `TRACE` and `NOTRACE`. They're useful when a program isn't behaving as intended and you need to find out why.

7.3.1 The `TRACE` Command

```
TRACE
```

`TRACE` displays line numbers as they are executed

`TRACE` causes Applesoft to display the line number of each statement it executes. Each line number displayed is preceded on the screen by a number sign (#). For example, the program

```
10 TRACE
20 I = I + 1 : I = I + 1
                          —two statements on line 20
30 J = J + 1 : J = J + 1
                          —two statements on line 30
40 GOTO 20                —loop forever
```

will display the output

```
#20 #20 #30 #30 #40 #20 #20 #30 #30 #40
#20 #20 #30 #30 #40 #20 #20 #30 #30 #40
#20 #20 #30 #30 #40 #20 #20 #30 #30 #40
#20 #20 #30 #30 #40 #20 #20 #30 #30 #40
#20 #20 #30 #30 #40 #20 #20 #30 #30 #40
...
```

ad nauseam, or until you press CONTROL-C, whichever occurs first.

CONTROL-C: see Section 1.3.2

Once tracing has been started, it can be canceled only by

NOTRACE statement: see Section 7.3.2

- executing the NOTRACE statement
- restarting the system

Apple IIe Monitor program: see *Apple IIe Reference Manual*

- typing CONTROL-B to the Monitor program

As the example above shows, TRACE can be used from within a program as well as in immediate execution. A more realistic use in debugging would be to test for some error condition and turn on tracing only if the error condition holds:

Using TRACE from within a program

```
IF X > Y THEN TRACE      —trace if variable values are
                           wrong
```

Be sure to remove the TRACE statements from your program after you've found and exterminated the bug!

display formatting: see Section 5.2.4

When the program being traced contains display-formatting statements (VTAB, HTAB, TAB, semicolons, commas), line numbers displayed by TRACE may appear in a confused fashion or may be overwritten entirely.

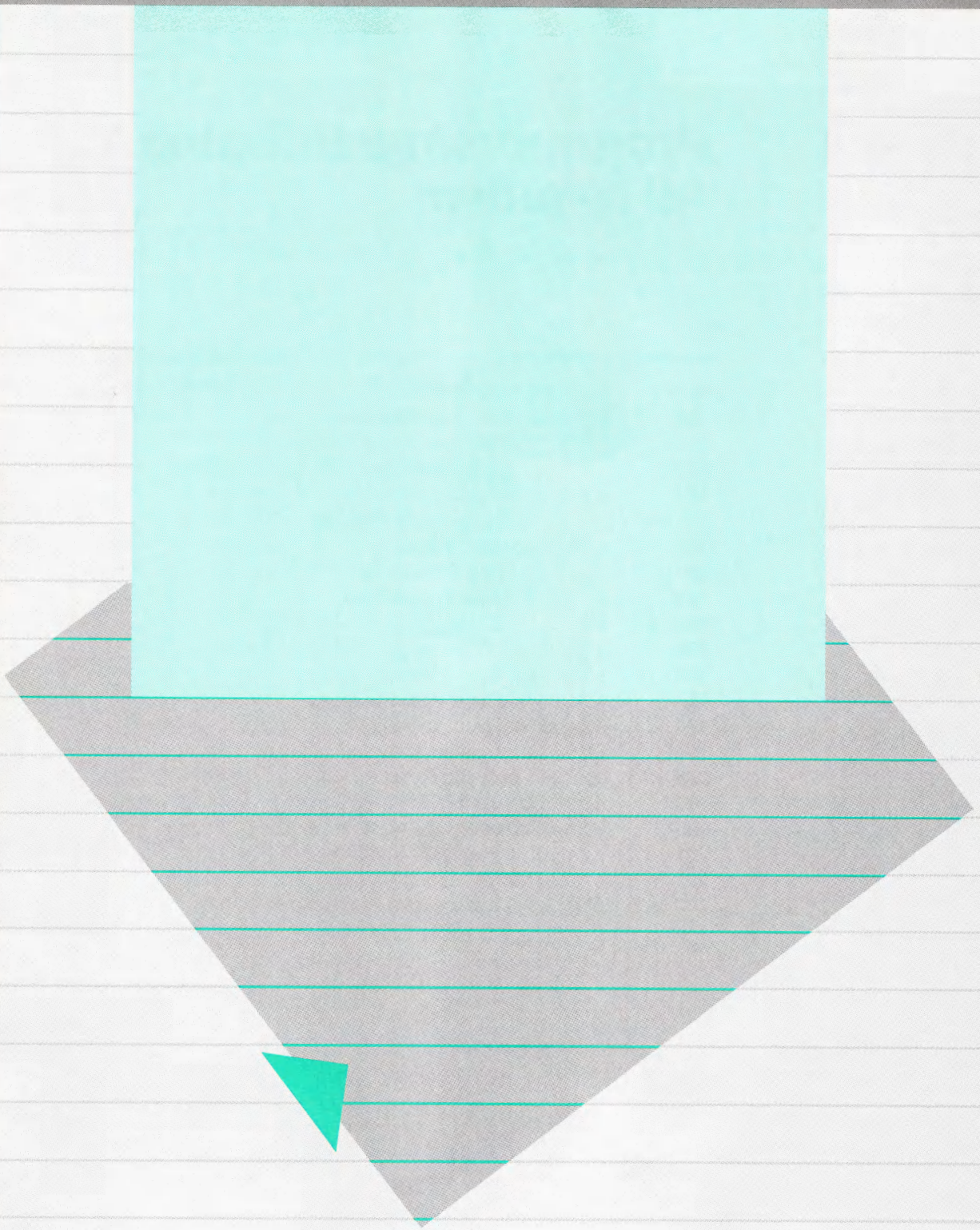
7.3.2 *The NOTRACE Command*

```
NOTRACE
```

The NOTRACE command cancels the effects of TRACE. After this command is executed, line numbers are no longer displayed on the screen as Applesoft executes them.

Chapter 8

Programming: Bringing It All Together



Programming: Bringing It All Together

Good programs don't just happen. Programs that are efficient, economical, and easy to debug and to modify are the result of careful planning. This chapter presents a method to facilitate such planning, using as its example a program to calculate postage fees for United States mail. A copy of this program is included on the Applesoft Sampler disk; a complete listing can be found in Appendix N.

program planning: see Section 8.1

Section 8.1, "Planning the Program," shows how to develop a list of program specifications and how to convert the list into a kind of program outline.

coding: see Section 8.2

Section 8.2, "Writing the Code," shows how to refine the outline developed in Section 8.1 into a final Applesoft program.

8.1 Planning the Program

Although you can afford to be "quick and dirty" for casual or one-time-only applications, you'll need to do some preliminary planning for more serious ones. In general, the more planning you do the more efficient and bug-free your finished program will be.

To demonstrate some of the principles of program planning, this chapter develops a program to calculate postage rates on certain classes of mail sent in the United States.

8.1.1 Program Specification

Good programs take **careful planning**

Program planning begins with deciding what your program is to do or what problem you want it to solve. You might want to design a space-war game, a tax planner, or a data-base management system. It doesn't matter how simple or how complex the task—whatever it is, you have to decide in detail what the program is supposed to do.

Specifications define what the program does and how

To make writing the program easier, it's a good idea to begin with a list of program *specifications*. This list specifies what information the

program needs, what the program should and should not do, how the results are to be presented, and so forth.

What the Program Needs

It's fairly simple to determine what information a postage rates program needs. Since the goal is to determine how much it costs to mail an item, and since cost is a function of mail class and weight, the program needs someone or something to tell it what weight item is being mailed in what class. To keep things simple, the program will get this information from the program user:

What does the program need **from the user?**

- The user tells the program the class of mail.
- The user also tells the program the weight of the item, since postage rates are based on both class and weight.

When the program has the weight and class of mail (for instance, three ounces of first class), it needs to determine the postage based on some scale. It must either calculate the postage with a formula or look up the rate in a table stored in the computer's memory. Since all computers are inherently stupid and must be told everything, you have to include formulas or tables with which your computer can work:

What **internal information** does the program need?

- The program includes formulas and/or tables of postage rates for various weights and classes of mail.
- The program includes information on the maximum allowable weight in each class.

This last specification is a matter of postal regulations; first class mail above 12 ounces is called priority mail and is charged at a different rate; 70 pounds is the maximum weight for priority mail; and so forth. Program planning, then, calls for information often outside of the programmer's immediate purview. That's why God created libraries and telephones—so that programmers could obtain information they don't already have.

What the Program Will and Won't Do

What **won't** the program do?

Deciding on the limits of a program is often as important as determining what the program is supposed to accomplish. United States mail has four classes, several types within certain classes, optional extras like insurance and various forms of registry, and so forth. To make designing and writing the program simpler, we'll assume that our postage is never below first class, and further that we never insure or register mail (what fools these mortals . . .). We'll also assume that packages sent by overnight delivery (express mail) never weigh more

than nine pounds and always travel in the same postal zone:

- The program is limited to express, first class, and priority mail (one zone only).
- The heaviest express mail package will be nine pounds; first class and priority mail may be of any weight, up to Post Office limits.

Validating the Data

Now that the program has information both from the user and from its own internal resources (charts of rates and so on), it must check the validity and consistency of the information.

First for *validity:* does the information the user typed make sense in terms of what the program expected? If the program needs a digit for the weight of a letter, what should it do if it gets a word? In designing any program, it's important to remember:

- Most humans do not possess genetic information about what to type into computers.

Humans will be human...

- Most humans make mistakes.

Give your user **clear instructions**

The program, then, must display clear instructions telling the user what to type (kind of information needed) and what form to use (letters, digits, words):

- The program will display a list of classes of mail on the screen, with instructions for the user about what to type.
- After it gets the class of mail, the program will solicit the weight from the user with proper instructions.
- There will be a mechanism for accepting valid information and rejecting invalid information (that is, there are error-handling provisions).

Naturally, if the program rejects invalid information, it must try again to get valid information from the user:

What if the information is **invalid** or **inconsistent?**

- If information is rejected as invalid, the program will continue to solicit information until it gets what it needs.

Now to consistency: although a user might plausibly ask the cost of sending a five-pound package via first class mail, only the program knows (by checking its table of limits) that five pounds is too heavy for first class. It must notify the user that some other action is called for:

- If information is rejected as inconsistent, the program will notify the user with appropriate recommendations for further action.

Displaying the Results

How are the **results** presented?

The specifications must also include the form in which the results are to be given to the user. In this case we'll keep it simple:

- The final calculated postage charge will be displayed on the screen with appropriate labeling.

What happens when the program is **finished?**

Finally, the specifications must tell what the program does when its job is completed. Here, it will repeat the whole process until the user types in some kind of "I'm done" signal:

- The program will continue to solicit information to calculate new postage charges until the user types an "end" signal.

Reordering the list of specifications into a more logical form, we obtain the final list shown in Table 8-1.

Table 8-1 Final Specifications for the Postage Rates Program

- The program will display a list of classes of mail on the screen, with instructions for the user about what to type.
- The program is limited to express, first class, and priority mail (one zone only).
- The user tells the program the class of mail.
- After it gets the class of mail, the program will solicit the weight from the user with proper instructions.
- The user tells the program the weight of the item.
- The program includes information on the maximum allowable weight.
- The heaviest express mail package will be nine pounds; first class and priority mail may be of any weight, up to Post Office limits.
- There will be a mechanism for accepting valid information and rejecting invalid information.
- If information is rejected as invalid, the program will continue to solicit information until it gets what it needs.
- If information is rejected as inconsistent, the program will notify the user with appropriate recommendations for further action.
- The program includes formulas and/or tables of postage rates for various weights and classes of mail.
- The final calculated postage charge will be displayed on the screen with appropriate labeling.
- The program will continue to solicit information to calculate new postage charges until the user types an "end" signal.

Reviewing the list, you can see that the program's actions fall into a natural chronological order:

1. Computer displays prompting messages.
2. User responds.
3. Program checks validity of responses.
4. If any information is invalid, program solicits new information.
5. Program checks consistency of responses.
6. If any information is inconsistent, program solicits clarified information.
7. Program processes validated information.
8. Program displays results and goes back to stage 1.

interactive program: a program that conducts a dialog with the user

You'll find that most *interactive* programs—programs that carry on a "dialog" with a human sitting at the computer—involve most of the categories above in roughly the same order.

8.1.2 Program Layout

Lay out your program before you start coding

Before rushing to put fingers to keyboard, it's best to take your planning at least one step further. Now is the time for program layout. Here you plan out the form for each section of the program as described in both the specification list and the chronological order list.

stepwise refinement: a technique of program development in which broad sections of the program are laid out first, then elaborated step by step

The program layout technique presented here is called *stepwise refinement.* What this means is laying out broad sections of the program, then going back and refining each section step by step.

The Initial Layout

Table 8-2 Initial Layout of the Postage Rates Program

Repeat

Display menu

Accept class

Accept weight

Compute charge

Display results

until *done*

Table 8-2 shows an initial layout of the Postage Rates program in the broadest terms. The layout says that there are five general sections to the program (*Display menu, Accept class, Accept weight, Compute charge,* and *Display results*), and that the program is to repeat this sequence of steps in order until somehow told to stop.

Each section can now be treated as an independent module, to be designed and coded separately. The smaller the chunks of program you work with and the more independent each chunk is, the less chance for error and the easier the program will be to debug.

Refining the Layout

Refine each module

Now that we have the program laid out in skeleton form, we can begin to put some flesh on the bones. Table 8-3 shows the first refinement, in which each of the broad steps in the initial layout is spelled out in more detail.

Table 8-3 First Refinement of the Postage Rates Program

```
Repeat

    Display menu:
        List choices

    Accept class:
        Instruct user how to choose
        Repeat
            Get postage class from user
        until valid menu item

    Accept weight:
        Repeat
            Instruct user how to type
            Get weight from user
        until consistent

    Compute charge:
        Calculate from formula or look up in table

    Display result:
        Format result with dollar sign, trailing zeros
        Label and display result
        Wait for signal from user before proceeding

until user signals end
```

At this point, many programmers would take outline in hand and attack the keyboard. (With an outline?) But a couple of the modules need further refinement: both the *Accept weight* and the *Compute charge* modules need to do specialized processing depending on the class of mail specified by the user. The new information in the *Compute charge* module comes from examining postage rate charts. First class mail is fairly regular, so a formula can be used to compute the charge. Express mail follows no regular pattern, so it's easier to create a table of charges. Priority mail requires a combination of both formula and table. The final program layout is shown in Table 8-4.

Table 8-4 Final Layout of the Postage Rates Program

```
Repeat

    Display menu:
        List choices

    Accept class:
        Instruct user how to choose
        Repeat
            Get postage class from user
        until valid menu item

    Accept weight:
        Repeat
            Instruct user how to type
            Get weight from user:
                Check validity of response
                Express?
                    If item more than 9 pounds
                        then suggest alternative
                First class?
                    If item more than 12 ounces
                        then suggest alternative
                Priority?
                    If item less than 12 ounces
                        then suggest alternative
                    If item more than 70 pounds
                        then suggest alternative
        until valid and consistent

    Compute charge:
        Express?
            Look up charge in table
        First class?
            Calculate charge from formula
        Priority?
            If item less than 10 pounds
                then look up charge in table
                otherwise calculate charge from formula

    Display result:
        Format result with dollar sign, trailing zeros
        Label and display result
        Wait for signal from user before proceeding

until user signals end
```

8.2 Writing the Code

Use the layout as a guide while writing code

Now that you've refined the program layout to a sufficient level of detail, you're finally ready to start writing code. The layout is only a guide; it isn't the last word. As you write and test the actual program, you may find you need to make changes in your original design. That's perfectly all right; use the layout to keep you on track.

Methodical program development makes programs easy to debug and modify

What follows in this section represents one way to turn the outline into a working program. It isn't the only way—a hundred programmers would produce a hundred different programs for the same task. It does, however, work; and because it's been developed in an orderly, methodical way, it's also logically organized and easy for a human reader to follow. This is an important consideration, because it makes the program easy to debug and easy to modify. (Almost all serious programs need to be modified at some time or other, often by someone other than the original programmer.)

The author makes no warranties, either express or implied...

Hysterical Note: Any resemblance between the following program and true top-down structured code is purely coincidental and probably hallucinatory. The perceiver of such a resemblance is advised to seek psychiatric aid promptly.

8.2.1 Preliminaries

Your program should begin with a block of REM statements identifying the program and describing what it does. Most programmers add their own name and the date of the program's current version:

```
10 REM POSTAGE RATES
                          —name of program
20 :                      —colon leaves line empty
30 REM DETERMINES POSTAGE FEES
                          —what program does
40 REM FOR EXPRESS, 1ST CLASS,
50 REM AND PRIORITY MAIL
                          —empty line inserted by embedding CONTROL-J (line feed) at end of REM statement in line 50

60 REM   V2   9/01/82
                          —number and date of this version
70 REM BY JOHN SCRIBBLEMONGER
                          —programmer's credit line
```

8.2.2 *Display the Menu*

Now you can refer to your outline and base your code directly on it. Notice the REM statements introducing the different sections. All the comments marked here by dashes (—) could also be included as REM statements:

```
100 REM MENU OF POSTAGE CLASSES
                              —CONTROL-J here
110 HOME                      —begin with a clear screen
120 TITLE$ = "POSTAGE RATES"
130 PRINT
140 HTAB 21 - LEN (TITLE$) / 2
                              —formula to center title
150 PRINT TITLE$
160 VTAB 6
170 PRINT "1. EXPRESS"
180 PRINT "2. FIRST CLASS"
190 PRINT "3. PRIORITY"
200 PRINT
210 PRINT "4. END THE PROGRAM"
                              —the escape hatch
```

8.2.3 *What's The Postage Class?*

This section finds out what mail class the user wants to use. Note the use of CONTROL-J, the line feed character, to set off the REM statements for easier reading (line 300):

```
300 REM                       —CONTROL-J here
       GET CLASS OF MAIL
                              —CONTROL-J here
310 VTAB 14
320 PRINT "Press the number of your
      choice:";               —semicolon keeps response on
                               same line
330 GET C$                    —only one keypress needed;
                               cuts down on error possibili-
                               ties. Note use of string variable
                               to get number; avoids type
                               mismatch errors
335 REM                       —CONTROL-J here
       CHECK FOR VALIDITY
                              —another CONTROL-J (last
                               time this is noted)
```

```
340 IF C$ = "4" THEN END
                          —end program if user types a 4
350 IF VAL (C$) > 0 AND VAL (C$) < 4
      THEN 380            —skip next two lines if valid
                           choice typed
360 PRINT CHR$(7); CHR$(7);
                          —beep twice to get attention
370 GOTO 330              —response was invalid; try
                           again
380 PRINT C$              —since choice accepted via
                           GET, it isn't displayed on the
                           screen. Display it back to user
390 C = VAL (C$)          —need this value later to deter-
                           mine what section of program
                           to branch to for proper
                           processing
```

8.2.4 *What Does It Weigh?*

Now the program asks the user for the weight of the letter or package. The program makes sure that the user follows the instructions and types a number for the weight and a symbol (O or P) for ounces or pounds. Notice that the program accepts both the numeric weight of the item and the ounce/pound designation in the same string (line 530).

```
500 REM
      GET WEIGHT OF ITEM
505 VTAB 16
510 PRINT "Please enter the WEIGHT - a
      number plus an O (for ounces) or a P
      (for pounds) - and press the RETURN
      key: ";             —prompting message to tell
                           user what information to type
                           and how to type it
520 CALL -868             —clear to end of line; useful to
                           erase any errors that might be
                           typed
530 INPUT ""; W$          —semicolon suppresses ques-
                           tion mark
540 W1$ = RIGHT$ (W$,1)
                          —rightmost letter should be
                           either O or P; use it later to see
                           if weight is consistent with
                           postal regulations
```

```
550 W = VAL (W$)          —how many ounces or pounds?
555 REM
       WAS ENTERED WEIGHT VALID?
560 IF W > 0 AND (W1$="O" OR W1$="P")
      THEN 710            —if a weight was typed, and if
                           last character was either O for
                           ounces or P for pounds, then
                           proceed
570 PRINT CHR$ (7); CHR$ (7)
                          —beep twice to get attention
580 GOTO 500              —entry was invalid; try again
```

If the program has progressed this far, then everything typed by the user is valid from the computer's point of view. However, the user's choices still may not be consistent with postal regulations or the program's limitations. First class letters must weigh less than 12 ounces, the program can't handle express mail above a certain weight, and so on. This section of code uses the value of variable C (set in line 390) to direct control to the proper subroutine to check for consistency.

```
700 REM
       CHECK CONSISTENCY
710 ON C GOSUB 10000, 11000, 12000
                          —branch to appropriate subrou-
                           tine to see if weight typed is
                           within postal rules or program
                           limitations for mail class
                           chosen
720 IF NOT EFLAG THEN 910
                          —if no inconsistency detected in
                           subroutine then proceed with
                           processing
730 GOSUB 60000 : REM KEYSTALL
                          —wait for user to acknowledge
                           message
740 EFLAG = 0             —clear error flag set in
                           subroutine
750 CLEAR                 —reset all variables, clear
                           arrays, etc.
760 GOTO 100              —restart program loop
```

8.2.5 *Compute the Charge*

Now that everything checks out all right, the program can proceed to calculate the postage. The calculation is different for each of the three classes of postage, so there are three separate calculating routines. Again, what routine the program goes to depends on the value of C, the number representing the postal class chosen by the user.

```
900 REM
       FIND APPROPRIATE CODE FOR
       PROCESSING          —everything is valid and consistent; now program can solve for the postage rate!
910 ON C GOSUB 1000, 2000, 3000
                           —branch to proper calculating routine
920 GOSUB 61000 : REM FORMATTER
                           —format result for display
930 PRINT
```

8.2.6 *Display the Results*

It's finally time to display the result!

```
935 REM
       DISPLAY RESULTS
940 PRINT "POSTAGE NEEDED: $"; T$
                           —finally, the postage due!
950 GOSUB 60000 : REM KEYSTALL
                           —don't go on until user is ready
960 CLEAR                  —prepare for restart...
970 GOTO 100               —...and do it.
```

8.2.7 *Calculating Routines*

The following three subroutines do the actual rate calculations, based on formulas, tables, or both. The rates for express mail are fairly straightforward; they are based on a table created in the express mail consistency-checking routine at line 10000. First class rates couldn't be simpler; a little arithmetic is all that's needed. Priority mail is another story, however; when you get to it, you'll find an explanation.

```
 999 REM
        SUBROUTINES BEGIN HERE
```

```
1000 REM
        EXPRESS MAIL CALCULATION
1010 W = INT (W + .99)
```
—weight must be increased to compensate for fractions; postal rates read "NOT MORE THAN *x* POUNDS"

```
1020 T = R (W)
```
—rate array filled in express mail consistency-checking routine (line 10000)

```
1030 RETURN
```
—end routine

```
2000 REM
        FIRST CLASS CALCULATION
2010 T = .20 + INT (W + .99 - 1) * .17
```
—first class rate is 20 cents first ounce plus 17 cents for each additional ounce or portion thereof (April, 1982 rates)

```
2020 RETURN
```
—end routine

Although there is something approaching a pattern to priority mail charges, the pattern is obscure at best. This is especially true for the first ten pounds. Pounds 1 through 5 are charged by the half-pound; pounds 6 through 10 are full-pound charges. It's simpler and quicker to use a table for these charges (lines 3030 to 3140) than to figure out a formula.

Weights over 5 pounds follow a more regular pattern than the first 5; they are all charged in full-pound increments. Furthermore, each five pounds costs $2.38. Unfortunately, the cost for pound 6 is different from the cost for pound 7, and so on. What it boils down to is that 5-pound lots can be charged at the same rate (line 3170), and anything that isn't a multiple of 5 must be looked up in a table (lines 3180 to 3220).

I'm not making this up...

If all this strains credulity, refer to United States Mail Service poster 103, November 1981.

```
3000 REM
        PRIORITY MAIL CALCULATION
3010 W = INT (W + .99)
```
—compensate for partial ounces or pounds

```
3020 IF W > 10 THEN 3160
                          —go to line 3160 for weights
                            greater than 10 pounds
                            (ounce weights converted to
                            pounds in consistency subrou-
                            tine starting at line 12000)
3025 REM
        PRIORITY RATES TO 10 POUNDS
3030 IF W <= 1 THEN T = 2.24
3040 IF W > 1 AND W <= 1.5 THEN
       T = 2.30           —rates in half-pound increments
3050 IF W > 1.5 AND W <= 2 THEN
       T = 2.54
3060 IF W > 2 AND W <= 2.5 THEN
       T = 2.78
3070 IF W > 2.5 AND W <= 3 THEN
       T = 3.01
3072 IF W > 3 AND W <= 3.5 THEN
       T = 3.25
3078 IF W > 3.5 AND W <= 4 THEN
       T = 3.49
3080 IF W > 4 AND W <= 4.5 THEN
       T = 3.73
3090 IF W > 4.5 AND W <= 5 THEN
       T = 3.97
3100 IF W > 5 AND W <= 6 THEN T = 4.44
                          —rates by the pound now!
3110 IF W > 6 AND W <= 7 THEN T = 4.92
3120 IF W > 7 AND W <= 8 THEN T = 5.39
3130 IF W > 8 AND W <= 9 THEN T = 5.87
3140 IF W > 9 THEN T = 6.35
3150 GOTO 3240            —branch to RETURN statement
3160 REM
        PRIORITY RATES FOR OVER 10 POUNDS
3170 T1 = INT (W / 5 - 1) * 2.38 + 3.97
                          —first 5 pounds cost $3.97; each
                            added 5 pounds cost $2.38
3180 W1 = W - INT (W / 5) * 5
                          —how many odd pounds are
                            there (pounds that are not
                            multiples of 5 and must be
                            charged at a special rate)?
3190 IF W1 = 1 THEN T2 = .47
3200 IF W1 = 2 THEN T2 = .95
3210 IF W1 = 3 THEN T2 = 1.42
```

```
3220 IF W1 = 4 THEN T2 = 1.90
3230 T = T1 + T2          —add the 5-pound-multiples rate
                             to the odd-pounds rate
3240 RETURN               —end routine
```

8.2.8 *Consistency-Checking Routines*

The next three routines make sure that first class letters aren't too heavy, that the requested rate can be calculated by the program, and in general that the program can deliver what the user wants. The express mail routine begins by loading its rates into a table (it gets the rates from a DATA list; DATA lists are excellent places to store information you might need in a program); then it checks to see if it has a rate for the package being sent. First class just makes sure that the package weighs 12 ounces or less; that's the maximum weight for a first-class item. Priority mail also has an easy job; it just makes sure the package weighs more than 12 ounces but not more than 70 pounds.

```
10000 REM
        EXPRESS MAIL CONSISTENCY CHECK
10010 DATA 9.35, 9.35, 9.55, 9.90,
        10.30, 10.65, 11.00, 11.40,
        11.75, 0          —express mail rates; 0 at end is
                             "last item" flag
10020 X = 0               —set up counter to check how
                             many rates are read from
                             DATA list
10030 X = X + 1           —increment counter
10040 READ R (X)          —put price into proper array
                             element
10050 IF R (X) = 0 THEN 10070
                          —price of 0 marks end of list
10060 GOTO 10030          —get next price
10070 X = X - 1           —X includes count of "last item"
                             flag from 10050; subtract it
                             from count since it's a
                             "dummy" item
10080 IF W1$ = "P" THEN 10100
                          —next line is for ounces only
10090 W = W / 16          —convert ounces to pounds
10100 IF W<=X THEN 10140
                          —if weight in pounds is covered
                             by the rate chart, then go
                             ahead
```

```
10110 PRINT
10120 PRINT CHR$ (7); CHR$ (7); "TOO
        HEAVY FOR MY TABLES - PLEASE CALL
        THE POST OFFICE"
                              —sorry; can't help you
10130 EFLAG = 1               —set flag indicating inconsistent
                               weight/type; will be checked at
                               line 720
10140 RETURN                  —end routine
11000 REM
         FIRST CLASS CONSISTENCY CHECK
11010 IF W1$ = "O" AND W < 12.01
        THEN 11060            —OK if not more than 12 ounces
11020 PRINT
11030 PRINT CHR$ (7); CHR$(7); "TOO
        HEAVY FOR FIRST CLASS"
                              —sorry—inconsistent!
11040 PRINT "TRY PRIORITY MAIL"
                              —suggest alternative
11050 EFLAG = 1               —set flag indicating inconsistent
                               weight/type; will be checked at
                               line 720
11060 RETURN                  —end routine
12000 REM
         PRIORITY MAIL CONSISTENCY CHECK
12010 IF W1$ = "P" THEN 12090
                              —if in pounds, then skip down
12020 IF W > 12 THEN 12080
                              —skip down if weight is between
                               12 and 16 ounces
12030 PRINT
12040 PRINT CHR$ (7); CHR$ (7);
        "TOO LIGHT FOR PRIORITY MAIL -"
                              —too light!
12050 PRINT "TRY FIRST CLASS"
                              —suggest alternative
12060 EFLAG = 1               —set flag indicating inconsistent
                               weight/type; will be checked at
                               line 720
12070 GOTO 12150              —branch to end of routine
12080 W = W / 16              —convert ounces to pounds
12090 IF W <= 70 THEN 12150
                              —final check: is item on the
                               charts?
```

```
12100 PRINT
12110 PRINT CHR$ (7); CHR$ (7);
        "TOO HEAVY FOR PRIORITY MAIL -"
                              —off the charts
12120 PRINT "TRY ONE OF THE AIR EXPRESS
        COMPANIES"            —too big for the Post Office!
12130 EFLAG = 1               —set flag indicating inconsistent
                               weight/type; will be checked at
                               line 720
12150 RETURN                  —end routine
```

8.2.9 *The "Keystall" Routine*

The "keystall" routine interrupts execution of the program and waits for the user to press a key before going on. The GET statement in line 60040 actually does the waiting; when the user presses a key, the program continues. What key the user presses doesn't matter—the program doesn't care what value is assigned to A$.

```
59999 REM
          UTILITY ROUTINES
                              —routines useful for various
                               tasks but ancillary to rest of
                               program

60000 REM
          KEYSTALL            —routine to interrupt program
                               until user presses a key

60010 VTAB 24                 —move cursor to screen bottom
60020 INVERSE                 —set text to appear black-on-
                               white
60030 PRINT "PRESS RETURN TO GO ON...";
60040 GET A$                  —wait for keypress
60050 NORMAL                  —restore ordinary white-on-
                               black
60060 RETURN                  —end routine
```

8.2.10 *The Formatting Routine*

After the postage charge is calculated, the program branches to this final subroutine. Here the final result is checked to see how it will look when it is displayed. Does it have a decimal point? Applesoft suppresses trailing zeros after a decimal point, but people are used to

seeing them when dealing with dollars and cents. The formatting subroutine adds trailing zeros as needed.

```
61000 REM
          MONEY FORMATTER
                              —adds zeros after the decimal
                                point where needed
61010 T$ = STR$ (T)           —turn the calculated postage
                                fee into a string
61020 IF T = INT (T) THEN T$ = T$ +
        ".00"                 —if charge is in whole dollars,
                                add a decimal point and two
                                zeros
61030 IF ASC (RIGHT$ (T$,2)) = 46 THEN
        T$ = T$ + "0"
                              —if second character from the
                                right is a decimal point (ASCII
                                code 46) then number has
                                only one digit to right of deci-
                                mal—so add a "0" to the
                                string
61040 RETURN                  —end the routine
```

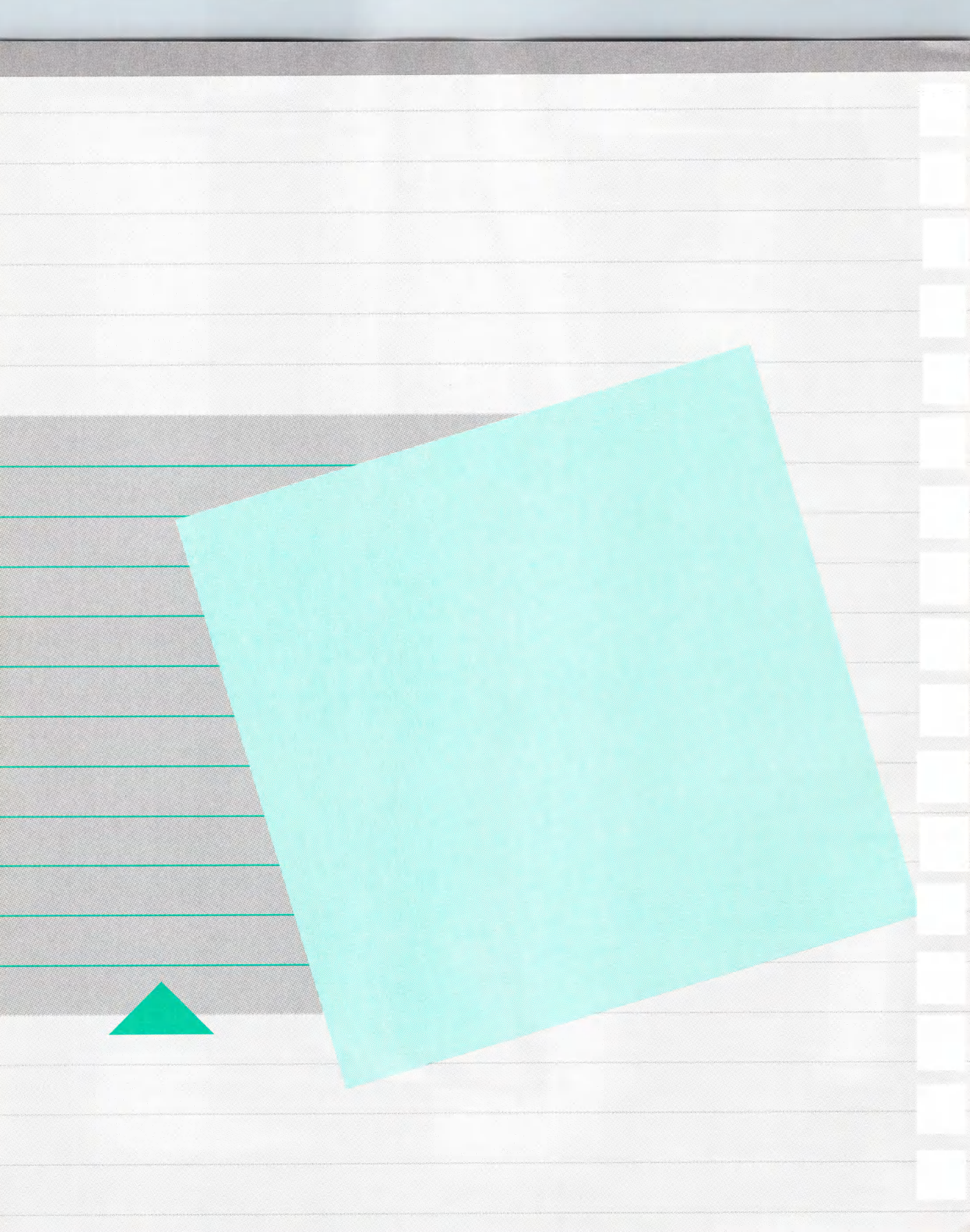
8.3 *Final Advice to the New Programmer*

Find your own style

The program planning methods discussed and demonstrated in this chapter won't necessarily work for everyone. Different people have different programming styles, and some people won't be comfortable with the (perhaps) coldly logical model presented here. What's important is to find a style that works for you. Programming is a logical art; it shouldn't be a confining one. Be as creative as your own internals will let you, remembering that poets also plan.

Keep in mind as you learn to program, please, when a bug is as hard to find as cheap gas, that deep down at the bit level—down where the computer deals with the only things it really understands—there are only zeros and ones.

(Funny—I don't *feel* comforted...)



Index

D

E

F

G

H

I

J

Q

R

S

Cast of Characters

Tuck end flap
inside back cover
when using manual.

Apple II

Applesoft BASIC Programmer's Reference Manual - Volume 1

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

030-0359-A